

RUST



Recopilado por Haylem Candelario Bauzá. Cuba 2025

One codebase, every platform.

Dioxus is *the* Rust framework for building fullstack web, desktop, and mobile apps. Iterate with live hotreloading, add server functions, and deploy in record time.

Introduction

Welcome to the Dioxus documentation! Dioxus is a framework for building cross-platform apps with the Rust programming language. With one codebase, you can build apps that run on web, desktop, and mobile.

Dioxus is designed to be familiar for developers who already know tools like React and Flutter.

```
use dioxus::prelude::*;
```

```
pub fn App() -> Element {  
    let mut count = use_signal(|| 0);  
  
    rsx! {  
        h1 { "High-Five counter: {count}" }  
        button { onclick: move |_| count += 1, "Up high!" }  
        button { onclick: move |_| count -= 1, "Down low!" }  
    }  
}
```



This guide is split into different sections:

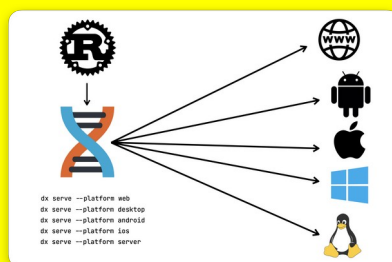
- [Tutorial](#) walks you through your first Dioxus app.
- [Core Concepts](#) provides detail on topics like managing state.
- [Guides](#) provides references for things like assets, routing, testing, and more.

First, try walking through the [Tutorial](#) to get familiar with Dioxus. Before embarking on a larger project, we strongly recommend reading the entire [Essential Concepts](#) and glancing through the [Guides Overview](#).

This guide assumes you already know some [Rust](#)! If not, we recommend reading [the book](#) to learn Rust first.

What is Dioxus?

Dioxus is a developer-friendly framework that empowers developers to ship cross-platform apps with one codebase.



In many ways, Dioxus is similar to Flutter: we integrate our own build tools, foster an ecosystem, and provide a markup language for declaring UI. In key areas, Dioxus takes a different approach:

- Apps are declared with HTML and CSS instead of custom styling solution
- Reactivity is inspired by web frameworks like React and SolidJS
- Dioxus code runs natively with no virtual machine and enables direct FFI with system APIs

Our goal is to provide a "better Flutter": faster, slimmer, and web-native. You can think of Dioxus as a hybrid of [Flutter](#) and [NextJS](#): cross-platform apps with stellar fullstack support. Today, Dioxus apps can only be written in Rust, but we plan to support more languages in the future.

Why Dioxus?

We started Dioxus because we believe the current standard of building apps is too complex. Developers need to learn and install dozens of different tools just to get their app into the world.



Our vision for Dioxus is a framework that is fast, flexible, and has a minimal learning curve. We want developers to confidently ship their app from idea to production as fast as possible. We believe that fewer tools and a simpler architecture makes it easier to develop apps. Apps that are easier to build also ship faster and are more likely to succeed.

Syntax and Ecosystem

The Dioxus syntax is similar to React's JSX markup, borrowing React's component and hooks approach. All components are Rust functions that take `Properties`, define state with hooks, and return an `Element`. We only support markup within the `rsx! {}` macro; this ensures your app is automatically optimized and has stellar devtools support like advanced hot-reloading.

```
#[component]
fn Component(name: String) -> Element {
    let mut count = use_signal(|| 0);

    rsx! {
        h1 { "Hello, {name}" }
        p { "Count: {count}" }
    }
}
```

Dioxus is designed to be easy to extend and fairly thin over system APIs. This means you can easily drop into system APIs when first-party APIs are lacking. When targeting the web, this might mean using [web-sys](#) and on Android using [jni](#).

```
fn PromptModal() {  
  #[cfg(web)]  
  web_sys::call_web_function();  
  
  #[cfg(android)]  
  jni_sys::call_android_function();  
}
```

The core Dioxus framework covers a number of utilities that are either challenging to design or integrate the dx tooling:

- [App Routing](#)
- [Backend integration via server functions](#)
- [Including and optimizing](#) assets
- [State management](#) (signals-based reactivity)
- [SDK](#): 1st-party System integrations

Stability

Dioxus has not reached a "1.0" release yet.

We are currently on version 0.6, which has stabilized a huge number of APIs and drastically improved the developer experience. In version 0.5 we overhauled the state management system and in 0.6 we overhauled tooling.

It's likely that the next few versions of Dioxus (0.7, 0.8) will bring breaking changes to your apps. Fortunately, these planned changes will only affect the syntax of specific APIs and not your apps at large. With every version update, we ship a rather comprehensive migration guide - eg [0.6](#).

Examples, projects, tutorials, and more

The Dioxus ecosystem is growing and so are the number of examples, projects, tutorials, books, and other learning resources.

We highly recommend a few first-party sources:

- The [official folder of small examples](#)
- The [official repository of example projects](#)
- The official [YouTube channel](#)

Who's funding Dioxus?

Dioxus is funded by a mix of corporate sponsorships, enterprise support contracts, [crowd-sourced funding](#), and [venture capital](#). We strive to maintain a healthy mix of funding to balance the various competing visions of the future. We want to provide a "Flutter but better" for everyone - not controlled by Apple, Meta, or Google - and we need to make sure Dioxus has a sustainable long-term financial future.

Ultimately, we'd like Dioxus to be self-sustaining. This means that you'll eventually have the option to deploy your production apps with [Dioxus Deploy](#). Revenue from *Dioxus Deploy* will in turn fund development on Dioxus itself.

We're committed to keeping Dioxus free and open source forever. You'll never need to pay us to build apps nor will we ever change the license of Dioxus.

Getting Started

Getting started with Dioxus is quick and shouldn't take more than a minute or two.

Pick an Editor

Pick your favorite editor!

We recommend using [VSCode](#) since Dioxus ships with its [own VSCode extension](#).

Our build tool `dx` is standalone and works with any editor.

Install Rust-Analyzer

Dioxus integrates very well with the [Rust-Analyzer LSP plugin](#) which provides appropriate syntax highlighting, code navigation, folding, and more.

You can follow the [installation instructions](#) for your editor of choice.

- [VSCode](#)
- [Zed](#)
- [Emacs](#)
- [Vim](#)

Install Rust

Head over to <https://rust-lang.org> and install the Rust compiler.

We strongly recommend going through the [official Rust book](#) *completely*. However, we hope that a Dioxus app can serve as a great first Rust project.

We've put a lot of care into making Dioxus syntax familiar and easy to understand, so you won't need deep knowledge of `async`, `lifetimes`, or `smart pointers` until you start building complex Dioxus apps.

Install cargo-binstall

Dioxus ships prebuilt binaries for its CLI using [cargo-binstall](#). This means you can install `dx` without needing to compile from source.

You can install `cargo-binstall` with the following command:

```
curl -L --proto '=https' --tlsv1.2 -sSf https://raw.githubusercontent.com/cargo-bins/cargo-binstall/main/install-from-binstall-release.sh | bash
```

or alternatively:

```
# through brew
brew install cargo-binstall

# or from source
cargo install cargo-binstall
```

Install the Dioxus CLI

Dioxus is comprised of two parts:

- The `dioxus` crate: the core Dioxus framework
- The `dx` tool: a CLI tool for building Dioxus apps

If you want to build web or mobile apps with Dioxus, or benefit from features like hot-reloading and bundling, you'll need to install the `dx` tool. You can download the prebuilt binary with the following command:

```
cargo binstall dioxus-cli
```

If you want to build the CLI from source, you can install it with the following command:

```
cargo install dioxus-cli
```

 Installing from source will take several minutes. We strongly recommend using `cargo-binstall` for installation.

If you get an OpenSSL error on installation, ensure the dependencies listed [here](#) are installed.

Platform-specific dependencies

Most platforms don't require any additional dependencies, but if you are targeting desktop, you can install the following dependencies.

macOS

There are no extra dependencies for macOS! However, if you want to build iOS apps, read the [iOS section](#) below.

Windows

Windows apps depend on WebView2 – a library that should be installed in all modern Windows distributions.

If you have Edge installed, then Dioxus will work fine. If you *don't* have WebView2, then you can [install it through Microsoft](#). Microsoft provides 3 options:

1. A tiny "evergreen" *bootstrapper* that fetches an installer from Microsoft's CDN.
2. A tiny *installer* that fetches WebView2 from Microsoft's CDN.
3. A statically linked version of WebView2 in your final binary for offline users.

We recommend using Option 1.

Linux

WebView Linux apps require WebkitGtk and xdotool. When distributing, these should be part of your dependency tree in your `.rpm` or `.deb`.

If you run into issues, make sure you have all the basics installed.

For Ubuntu, make sure everything is installed:

```
sudo apt update
sudo apt install libwebkit2gtk-4.1-dev \
  build-essential \
  curl \
  wget \
  file \
  libxdo-dev \
  libssl-dev \
  libayatana-appindicator3-dev \
  librsvg2-dev
```

For arch:

```
sudo pacman -Syu
sudo pacman -S --needed \
  webkit2gtk-4.1 \
  base-devel \
  curl \
  wget \
  file \
  openssl \
  appmenu-gtk-module \
  libappindicator-gtk3 \
  librsvg \
  xdotool
```

For all other Linux targets, [check the Tauri docs which cover the same dependencies](#).

In addition to the Tauri docs, for Fedora:

```
sudo dnf install libxdo-devel
```

WSL

While doable, it can be tricky to setup development in WSL for Dioxus desktop. Not everything has been figured out and some stuff may not work.

Here are the steps we used to get Dioxus running through WSL:

1. Update your kernel to the latest version and update WSL to version 2.
2. Add `export DISPLAY=:0` to `~/.zshrc`
3. Install Tauri's Linux dependencies found [here](#).
4. For file dialogs to work, you need to install a fallback like `zenity`

When running Dioxus desktop on WSL, you may get warnings from `libEGL`. There is currently no way to silence these, but the app should still render.

iOS

Building iOS apps requires a device running macOS with XCode installed.

Download and install XCode from one of the following places:

- [Mac App Store](#)
- [Apple Developer website](#)

You will need to download the iOS SDK and install some simulators.

For more details, we recommend reading the [dedicated guide for iOS development](#).

Android

Android apps require the Android SDK and NDK to be installed. This can be a substantial amount of setup, so we recommend reading the [dedicated guide for Android development](#).

Dioxus Tutorial

In this tutorial, we'll be building a small app called: *HotDog* - basically Tinder, but for dogs! This app will serve as a great way to learn about building UIs, adding state, and deploying.

By the end of this tutorial, you will launch your very own web, desktop, and mobile apps and a backend deployed to [Fly.io](#).



We will primarily focus on the higher-level concepts of Dioxus without diving deep into the details of specific APIs. We recommend experimenting with the APIs yourself or reading the [Core Concepts](#) and specific [Guides](#) for more information.

What will we be learning?

This guide will cover the "core" Dioxus features including:

- [Tooling Setup](#)
- [Creating a new app](#)
- [How Components Work](#)
- [Creating UI with RSX](#)
- [Styling and Assets](#)
- [Adding State](#)
- [Fetching Data](#)
- [Adding a Backend](#)
- [Integrating a Database](#)
- [App Routing](#)

- [Bundling](#)
- [Deployment](#)
- [Next Steps](#)

Dioxus is a very full-featured framework, so we encourage you to follow up this tutorial by building your own larger apps.

What are we building?

The features of *HotDog* are fairly simple:

- Engage with a stream of cute dog photos
- Swipe right if we want to save the dog photo to our collection
- Swipe left if we don't want to save the dog photo
- View the dog photos we saved later

At the end of the tutorial, you'll have your very own *HotDog* app to remix and download to your device.

Setting up Tooling

Before we get started, make sure you've followed the [Getting Started](#) page on installing the required dependencies.

We will be primarily developing *HotDog* as web application, but we still recommend setting up the relevant tooling for desktop and mobile development as well.

Checklist

We covered the setup instructions in [Getting Started](#), but first you should verify everything is set up properly:

- Rust is installed
- You have a code editor installed
- The wasm32-unknown-unknown Rust toolchain is installed
- The `dioxus -c li` is installed and up-to-date
- System-specific dependencies are installed

Verify your setup

Before proceeding, make sure you have the `dioxus -c li` installed and up-to-date.

Verify the returned version matches this guide (eg 0.6) by running:

```
dx --version
```

All the Commands

You can also run `dx help` which will give you a list of useful commands and some information on how to use `dx`.

Build, Bundle & Ship Dioxus Apps

Usage: dx [OPTIONS] <COMMAND>

Commands:

```
build      Build the Dioxus project and all of its assets
translate  Translate a source file into Dioxus code
serve      Build, watch & serve the Dioxus project and all of its assets
new        Create a new project for Dioxus
init       Init a new project for Dioxus in the current directory (by
default).  Will attempt to keep your project in a good state
clean      Clean output artifacts
bundle     Bundle the Dioxus app into a shippable object
fmt        Automatically format RSX
check      Check the project for any issues
run        Run the project without any hotreloading
config     Dioxus config file controls
help       Print this message or the help of the given subcommand(s)
```

Options:

```
--verbose    Use verbose output [default: false]
--trace      Use trace output [default: false]
--json-output Output logs in JSON format
-h, --help   Print help
-V, --version Print version
```

If dx is installed properly, then you're ready to proceed!

Create a new project

Let's get to work!

You can create a new Dioxus project by running the following command and following the prompts:

```
dx new hot_dog
```

You'll need to select a template to use to get started.

- Bare-bones: a very simple setup with just a `main.rs` and `assets` folder.
- Jumpstart: a scaffolded app with components, views, and suggested structure.
- Workspace: a full cargo workspace setup with different crates per platform.

We're going to use the bare-bones template for *HotDog*. Our app won't be too complex and can fit in one file.

- Select "false" when asked if you want to create a fullstack website.
- Select "false" for the router, though we *will* eventually add the router to the app.
- Select no for TailwindCSS. If you want to use Tailwind, make sure to read the [TailwindCSS guide](#).
- Select "Web" as the default platform.

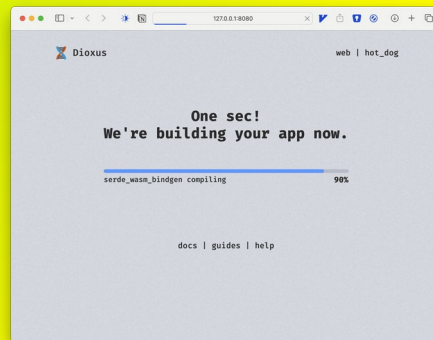
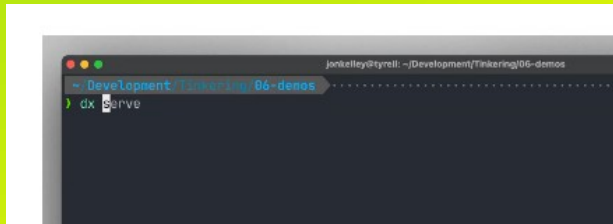
🔧 You don't need `dx new` to create new Dioxus apps! Dioxus apps are Rust projects and can also be built with tools like cargo.

Running the project

Once the project is generated, you can start it with the following command:

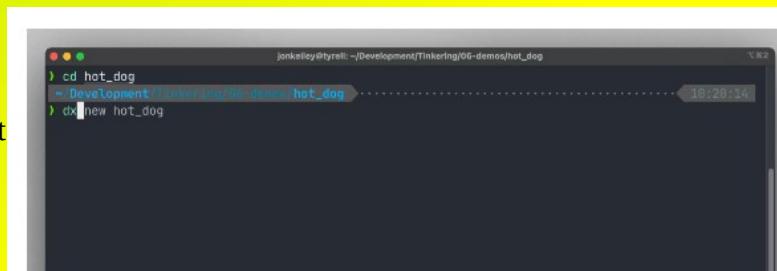
```
cd hot_dog
dx serve
```

This will start the cargo build and launch a web server to serve your app. If you visit the "serve" address (in this case, `http://127.0.0.1:8080`), then you'll receive a loading screen in your browser:



Once the app is loaded, you should be greeted with the default Dioxus template app:

Congrats! You have your very first Dioxus app.



Structure of the app

Open the app in your editor and take a look at its structure:



All Rust apps are comprised of a root `Cargo.toml` with a `main.rs` file located in the `src` folder. Our CLI `dx` pre-filled these files with the `dioxus` dependency and some starter code for us to get building quickly.

Assets in Dioxus can be placed anywhere in the project, but we suggest leaving them in the `assets` folder.

The Cargo.toml

The `Cargo.toml` outlines the dependencies to our app and specifies compiler settings. All Rust apps are *compiled*: we execute the Rust tool `cargo` which aggregates our `.rs` files together and generates a final binary executable (like a `.exe`) that runs our app.

All Dioxus apps will include `dioxus` as a dependency:

```
[dependencies]
dioxus = { version = "0.6.0" }
```

The prebuilt Dioxus templates initialize different cargo features for your app. `dx` will use these to decide which cargo features to enable when you specify the `--platform` feature. For example, if you use `dx serve --platform desktop` to build your app for desktop, `dx` will call `cargo build --no-default-features --features desktop`.

```
[features]
default = ["web"]
web = ["dioxus/web"]
desktop = ["dioxus/desktop"]
mobile = ["dioxus/mobile"]
```

Starting with Dioxus 0.6, `dx` will also initialize separate [Cargo profiles](#) for your app. These profiles let you customize the optimization level of each platform. `dx` also uses these platforms as a mechanism of isolating builds from each other.

Dioxus.toml

The `Dioxus.toml` file contains Dioxus-specific configuration for stages like bundling and deploying. Before Dioxus 0.5, we used the `Dioxus.toml` to specify asset inclusion and hot-reload watch paths, but as of Dioxus 0.6, these fields are deprecated and replaced by standards like `asset!()` and `.gitignore`.

We won't need to configure the `Dioxus.toml` for our app just yet.

Assets Folder

To include assets in your Dioxus app, you'll want to use the `asset!()` macro that we'll cover later in the [Styling and Assets](#) chapter. You can include assets from anywhere within your app's file tree, but we recommend using the pregenerated `assets` folder.

main.rs

Finally, the `main.rs`. The `main.rs` file is the entrypoint of our app, containing the `fn main` function. All Rust executables start their life at `main`.

The `main` of our HotDog app looks like this:

`src/guide_new_app.rs`

```
use dioxus::prelude::*;
```

```
fn main() {
    dioxus::launch(App);
}
```

The `launch` function calls the platform-specific `launch` function depending on which feature (web/desktop/mobile) is enabled on `dioxus`. `launch` accepts a root component, typically called `App`.

We'll cover components more in-depth in the [next chapter](#).

Resetting to Basics

The bare-bones template provides basic starter code for our app. However, we want to start *truly* from scratch, so we'll wipe away the `Hero` component and empty the `App` component to its basics:

src/guide_new_app.rs

```
use dioxus::prelude::*;

fn main() {
    dioxus::launch(App);
}

#[component]
fn App() -> Element {
    rsx! { "HotDog!" }
}
```

Your First Component

Now that we've initialized our *HotDog* app, we can start building out its components.

What is a component?

In Dioxus, apps are comprised of individual functions called *Components* that take in some *Properties* and render an *Element*:

src/guide_component.rs

```
fn DogApp(props: DogAppProps) -> Element {
    todo!()
}
```

Component Properties

All components take an object that outlines which parameters the component can accept. All `Props` structs in Dioxus need to derive the `Properties` trait which requires both `Clone` and `PartialEq`:

src/guide_component.rs

```
#[derive(Props, PartialEq, Clone)]
struct DogAppProps {
    breed: String,
```

```
}
```

Dioxus provides the `#[component]` macro for simplifying how components are defined. This macro converts the parameters of the annotated function into a hidden accompanying struct.

src/guide_component.rs

```
#[component]
fn DogApp(breed: String) -> Element {
    todo!()
}
```

When building apps, you'll frequently use the `#[component]` macro. When building libraries, we generally suggest deriving Props instead.

Properties are Immutable

If you're familiar with JavaScript, then you might also be familiar with libraries like [React](#). Dioxus is *very* similar to React: if you know React then you will feel comfortable with Dioxus.

Just like React, Dioxus components are rendered by calling the function `component`. On every render, Dioxus makes a `.clone()` of the component's props. This ensures you can't accidentally modify your props which can lead to hard-to-track issues with state management.

src/guide_component.rs

```
#[component]
fn DogApp(breed: String) -> Element {
    tracing::info!("Rendered with breed: {breed}");

    todo!()
}
```

Dioxus provides types that make `.clone()` cheaper to call, so don't worry about performance gotchas here.

Component Functions are Called Multiple Times

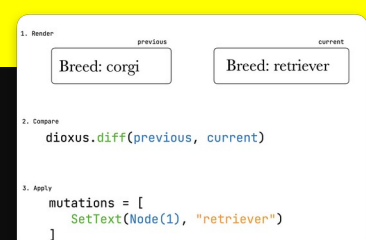
Just like React, Dioxus will call your component function multiple times throughout its lifecycle. This is called *re-rendering*. In Dioxus, re-renders are extremely cheap (much cheaper than React!). In most cases you shouldn't worry about re-rendering too frequently.

When Dioxus re-renders your component, it compares the `Element` returned from the *last* render against the `Element` returned in the *current* render.

For example, when the `breed` property changes on the `DogApp` component, Dioxus will call the `DogApp` function a second time and compare the previous `Element` against the new `Element`.

src/guide_component.rs

```
#[component]
fn DogApp(breed: String) -> Element {
    rsx! {
        "Breed: {breed}"
    }
}
```



Dioxus will re-render your component in only two circumstances:

- When the `Props` change as determined by `PartialEq`
- When a function like `signal.set()` or `signal.write()` calls `Scope.needs_update()`

Unlike React, all Dioxus components are *memoized by default* meaning Dioxus will always compare `Props` before deciding to re-render your component. As an additional optimization, Dioxus only compares dynamic parts of your `RSX`. Elements that don't contain dynamic data won't be checked for changes.

Composing Components

In Dioxus, *Components* are composed together to create *Apps*. Each component will hold onto its own state and handle its own updates. This makes it easy to abstract your app into different parts and even share pieces of your app as libraries for others to use.

To compose components together, we'll use the `rsx! {}` macro to define the structure of our app.

src/guide_component.rs

```
#[component]
fn App() -> Element {
    rsx! {
        Header {}
        DogApp { breed: "corgi" }
        Footer {}
    }
}
```

We'll cover `rsx! {}` in more depth in the [next chapter](#).

Describing the UI

Dioxus is a *declarative* framework. This means that instead of telling Dioxus what to do (e.g. to "create an element" or "set the color to red") we simply *declare* what we want the UI to look like using `RSX`.

src/hello_world_desktop.rs

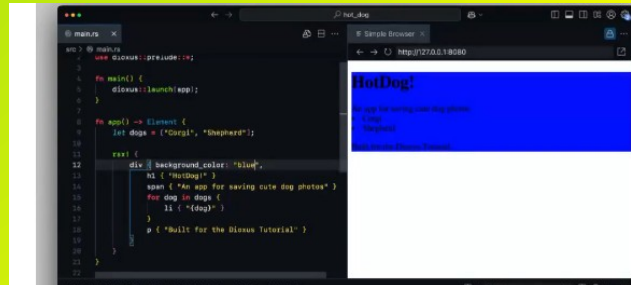
```
// define a component that renders a div with the text "Hello, world!"
fn App() -> Element {
    rsx! {
        div { "Hello, world!" }
    }
}
```

Here, we use the `rsx!` macro to *declare* that we want a `div` element, containing the text "Hello, world!". Dioxus takes the `RSX` and constructs a user interface from it.

Editing RSX with Hot-Reloading

When using `dx serve`, your app's RSX is automatically hot-reloaded whenever you edit and save the file. You can edit RSX structure, add new elements, and style your markup without a full rebuild.

Whenever you edit *Rust* code, then `dx` will automatically force a "full rebuild" of your app.



For an in-depth guide in what can and can't be hot-reloaded, check the [hot-reload guide](#) in the reference.

RSX is just HTML

Dioxus provides the `rsx! {}` macro for assembling `Elements` in your app. The `rsx! {}` macro primarily speaks HTML: the web, desktop, and mobile Dioxus first-party renderers all use HTML and CSS as the layout and styling technologies.

This means you can reuse your knowledge of the web and build your app using `div`, `span`, `img`, `style`, `button`, and more.

The RSX syntax is a "strict" form of Rust that uses Rust's `Struct` syntax for assembling elements:

src/guide_rsx.rs

```
rsx! {  
    div {  
        class: "bg-red-100"  
    }  
}
```

Elements in RSX differ slightly from Rust struct syntax: they can also contain child structs placed immediately after the final attribute.

src/guide_rsx.rs

```
rsx! {  
    div { class: "bg-red-100",  
        button {  
            onclick: move |_| info!("Clicked"),  
            "Click me!"  
        }  
    }  
}
```

Additionally, all quoted strings in JSX imply `format!("{}", ...)` automatically, so you can define a variable outside your markup and use it in your strings without an explicit format call:

src/guide_rsx.rs

```
rsx! {  
    div { "Breed: {breed}" }  
}
```

Any expression that can be rendered to a `String` can be included directly in JSX. JSX also accepts `Option<Element>` and iterators of `Elements`:

src/guide_rsx.rs

```
rsx! {  
    // Anything that's `Display`  
    {"Something"}  
  
    // Optionals  
    {show_title.then(|| rsx! { "title!" } )}  
  
    // And iterators  
    ul {  
        {(0..5).map(|i| rsx! { "{i}" })}  
    }  
}
```

Dioxus provides two items of syntax sugar for these common cases: `for` loops and `if` chains. These blocks return the contained JSX directly.

src/guide_rsx.rs

```
rsx! {  
    if show_title {  
        "title!"  
    }  
  
    ul {  
        for item in 0..5 {  
            "{item}"  
        }  
    }  
}
```

For lists, Dioxus uses the `key` attribute to ensure it's comparing the right elements between renders. If you forget to add a `key` attribute to your list item, you might run into performance and state management issues. Usually you can find a unique key to differentiate your list items:

src/guide_rsx.rs

```
rsx! {  
    for user in users {  
        div {  
            key: "{user.id}",  
            "{user.name}"  
        }  
    }  
}
```

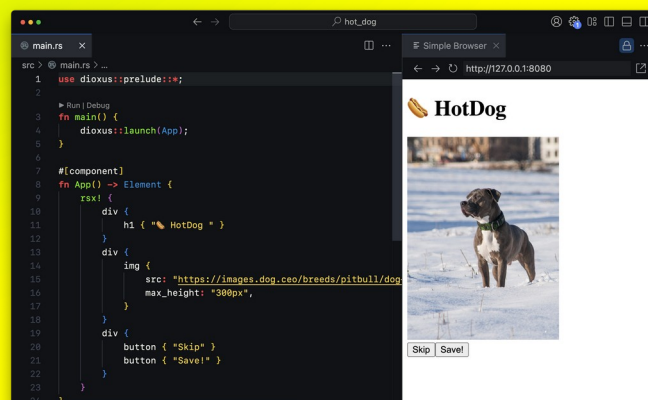
Adding UI to our *HotDog* App

Let's add a basic UI to our app. We'll add a header, a body image for the dog photo, and some basic buttons.

src/guide_rsx.rs

```
#[component]
fn App() -> Element {
  rsx! {
    div { id: "title",
      h1 { "HotDog! 🌭" }
    }
    div { id: "dogview",
      img { src: "https://images.dog.ceo/breeds/pitbull/dog-3981540_1280.jpg" }
    }
    div { id: "buttons",
      button { id: "skip", "skip" }
      button { id: "save", "save!" }
    }
  }
}
```

Our app is coming together!



Unfortunately, it's not very beautiful yet. Let's move on to styling our app.

Styling and Assets

Unfortunately, our HotDog app isn't quite ready to show off - it's completely unstyled!

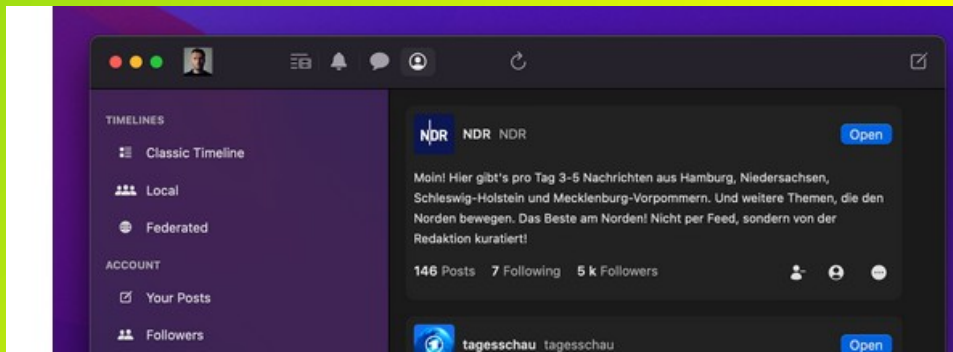
In this chapter we'll cover adding assets and styles to our app.

Dioxus uses CSS for Styling

As mentioned earlier, Dioxus apps use HTML and CSS as the core markup and styling technology. Instead of re-inventing the wheel like Flutter and React-Native, we designed Dioxus to use HTML and CSS on every platform.

CSS is by-far the most popular styling system and is extremely capable. For example, here's a screenshot of ebou, a very beautiful Mastodon client built with Dioxus.

HTML and CSS are very powerful - don't worry about being too limited!



Adding the CSS File with asset!()

The bare-bones template already includes a base `main.css` in the `assets` folder.

```
├── Cargo.toml
├── assets
│   └── main.css
├── src
│   └── main.rs
```

To include the CSS in our app, we can use the `asset!()` macro. This macro ensures the asset will be included in the final app bundle.

`src/guide_assets.rs`

```
static CSS: Asset = asset!("/assets/main.css");
```

We also need to load the asset into our app using the `document::Stylesheet` component. This component is equivalent to the `<link>` HTML element but also ensures the CSS will be pre-loaded during server-side-rendering.

`src/guide_assets.rs`

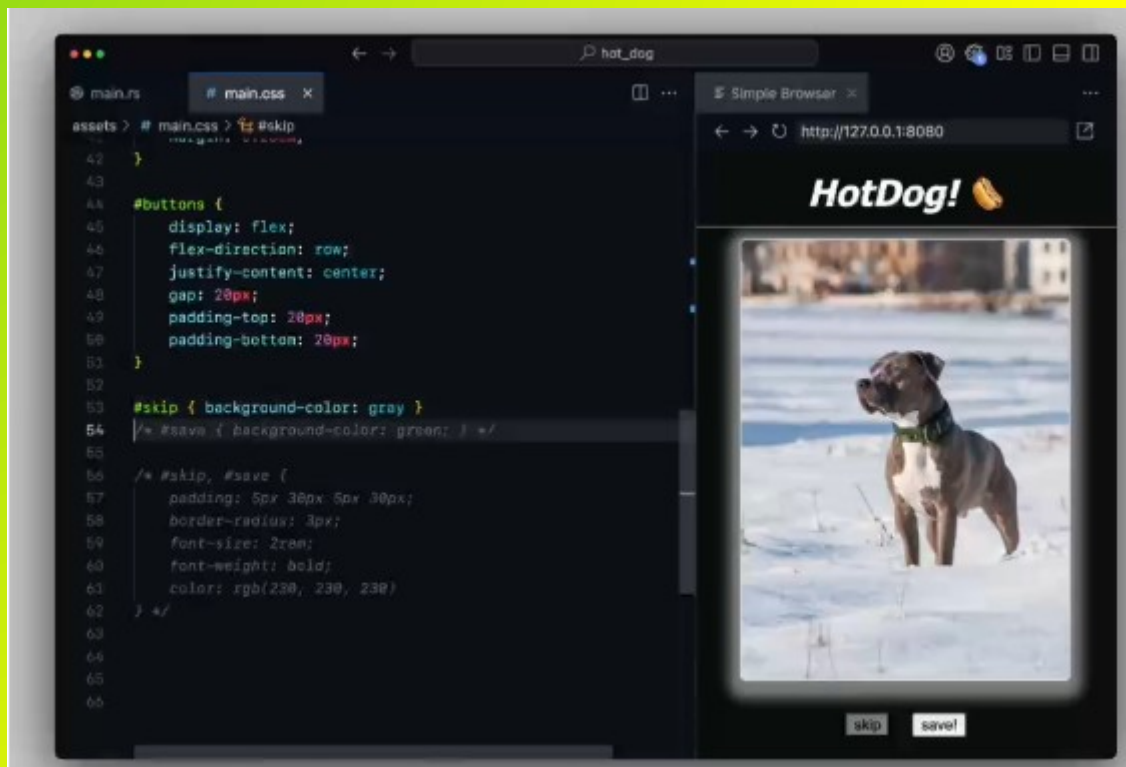
```
fn App() -> Element {
  rsx! {
    document::Stylesheet { href: CSS }
  }
}
```

Unlike Rust's `include_str!()` macro, the `asset!()` macro does not actually include the *contents* of the asset in our final executable. Instead, it generates a unique path so that the asset can be loaded at runtime. This is ideal for web apps where assets are loaded in parallel through different HTTP requests.

🔑 The `asset!()` macro generates a unique name that won't exactly match the input name. This helps prevent name collisions and improves caching.

Hot-Reloading

All assets in Dioxus participate in hot-reloading. Try editing your app's `main.css` and watch changes propagate in real time.



Including Images

In Dioxus, you can include images in two ways:

- Dynamically with a URL
- Statically with the `asset!()` macro.

When including assets with a URL, simply fill the `src` attribute of `img {}`. Note that when the app is offline, URL-based images won't download.

src/guide_assets.rs

```
rsx! {
  // ...
  div {
    img { src: "https://images.dog.ceo/breeds/pitbull/dog-3981540_1280.jpg" }
  }
}
```

For static images, you can use the same `asset!()` macro that we used to include the app's CSS.

src/guide_assets.rs

```
static ICON: Asset = asset!("/assets/icon.png");

rsx! {
  img { src: ICON }
}
```

Optimizations

By default, the `asset!( )` macro will lightly optimize CSS, JavaScript, JSON, and images. The name of the asset will also be modified to include a content hash.

src/guide_assets.rs

```
// would output main-j1238nask123.css
asset!("/assets/main.css").to_string();
```

You can optimize assets even further, with an optional `Options` struct. For example, `dx` can automatically convert `.png` images to a more optimized `.avif` format:

src/guide_assets.rs

```
// outputs icon-j1238jd2.avif
asset!("/assets/icon.png", ImageAssetOptions::new().with_avif());
```

For many apps, asset optimization is the most effective way of improving load times. As developers, we frequently overlook the size of images and accidentally make our sites load slower.

Check out the [assets guide](#) for a more in-depth explanation of how the Dioxus asset system works.

The Final CSS

We can use the asset hot-reload system of `dx` and our knowledge of CSS to create a beautiful app:



The final CSS is here for reference:

```
/* App-wide styling */
html, body {
  background-color: #0e0e0e;
  color: white;
  font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
  height: 100%;
  width: 100%;
  overflow: hidden;
  margin: 0;
}

#main {
  display: flex;
  flex-direction: column;
  height: 100%;
  justify-content: space-between;
```

```

}

#dogview {
  max-height: 80vh;
  flex-grow: 1;
  width: 100%;
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
}

#dogview img {
  display: block;
  max-width: 50%;
  max-height: 50%;
  transform: scale(1.8);
  border-radius: 5px;
  border: 1px solid rgb(233, 233, 233);
  box-shadow: 0px 0px 5px 1px rgb(216, 216, 216, 0.5);
}

#title {
  text-align: center;
  padding-top: 10px;
  border-bottom: 1px solid #a8a8a8;
  display: flex;
  flex-direction: row;
  justify-content: space-around;
  align-items: center;
}

#title a {
  text-decoration: none;
  color: white;
}

a#heart {
  background-color: white;
  color: red;
  padding: 5px;
  border-radius: 5px;
}

#title span {
  width: 20px;
}

#title h1 {
  margin: 0.25em;
  font-style: italic;
}

#buttons {
  display: flex;
  flex-direction: row;
  justify-content: center;
  gap: 20px;
  /* padding-top: 20px; */
  padding-bottom: 20px;
}

#skip { background-color: gray }
#save { background-color: green; }

```

```
#skip, #save {
  padding: 5px 30px 5px 30px;
  border-radius: 3px;
  font-size: 2rem;
  font-weight: bold;
  color: rgb(230, 230, 230)
}

#navbar {
  border: 1px solid rgb(233, 233, 233);
  border-width: 1px 0px 0px 0px;
  display: flex;
  flex-direction: row;
  justify-content: space-around;
  padding: 20px;
  gap: 20px;
}

#navbar a {
  background-color: #a8a8a8;
  border-radius: 5px;
  border: 1px solid black;
  text-decoration: none;
  color: black;
  padding: 10px 30px 10px 30px;
}

#favorites {
  flex-grow: 1;
  overflow: hidden;
  display: flex;
  flex-direction: column;
  padding: 10px;
}

#favorites-container {
  overflow-y: auto;
  overflow-x: hidden;
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
  justify-content: center;
  gap: 10px;
  padding: 10px;
}

.favorite-dog {
  max-height: 180px;
  max-width: 60%;
  position: relative;
}

.favorite-dog img {
  max-height: 150px;
  border-radius: 5px;
  margin: 5px;
}

.favorite-dog: hover button {
  display: block;
}

.favorite-dog button {
```

```
display: none;
position: absolute;
bottom: 10px;
left: 10px;
z-index: 10;
}
```

Interactivity

Now that our *HotDog* app is scaffolded and styled, we can finally add some interactive elements.

Encapsulating State

Before we get too far, let's split our app into two parts: the `Title` and the `DogView`. This will help us organize our app and keep the `DogView` state separated from `Title` state.

src/guide_state.rs

```
#[component]
fn App() -> Element {
  rsx! {
    document::Stylesheet { href: CSS }
    Title {}
    DogView {}
  }
}

#[component]
fn Title() -> Element {
  rsx! {
    div { id: "title",
      h1 { "HotDog! 🐕" }
    }
  }
}

#[component]
fn DogView() -> Element {
  rsx! {
    div { id: "dogview",
      img { src: "https://images.dog.ceo/breeds/pitbull/dog-3981540_1280.jpg" }
    }
    div { id: "buttons",
      button { id: "skip", "skip" }
      button { id: "save", "save!" }
    }
  }
}
```

Event Handlers

In the `DogView` component, we want to attach an action to the click of the buttons. For example: skipping or saving the current dog photo. We can use an [EventHandler](#) to listen for the `click` events.

Event handlers are similar to regular attributes, but their name usually starts with `on` - and they accept closures as values. The closure will be called whenever its corresponding event is triggered. The listener receives information about the event in the [Event](#) object.

We'll add some closures inline and then pass them to the `onclick` attribute for both the *skip* and *save* buttons:

src/guide_state.rs

```
#[component]
fn DogView() -> Element {
    let skip = move |evt| {};
    let save = move |evt| {};

    rsx! {
        // ...
        div { id: "buttons",
            button { onclick: skip, id: "skip", "skip" }
            button { onclick: save, id: "save", "save!" }
        }
    }
}
```

You can read more about Event Handlers in the [Event Handler reference](#)

State with use_hook

So far, our components have no internal state. For our `DogView`, we want to change the currently displayed dog photo whenever the user clicks *skip* or *save*.

To store state in components, Dioxus provides the `use_hook` function. This makes it possible for bare Rust functions to store and load state without the use of an extra struct.

When called in a component, the `use_hook` function will return a `.clone()` of the originally stored value:

src/guide_state.rs

```
#[component]
fn DogView() -> Element {
    let img_src = use_hook(|| "https://images.dog.ceo/breeds/pitbull/dog-3981540_1280.jpg");

    // ..

    rsx! {
        div { id: "dogview",
            img { src: "{img_src}" }
        }
        // ..
    }
}
```

Dioxus hooks are very similar to React's hooks and need to follow some [simple rules](#) to function properly.

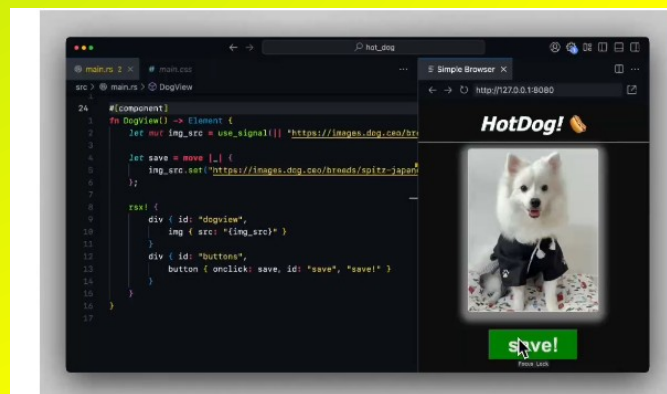
Signals and use_signal

While `use_hook` makes it possible to store any value that implements `Clone`, you'll frequently want a more capable form of state management. Built-in to Dioxus are *signals*.

`Signal` is a wrapper type around an ordinary Rust value that tracks reads and writes, bringing your app to life. You can wrap any Rust value in a signal. Signals can be created manually with `Signal::new()` but we strongly recommend using the `use_signal` hook instead.

🔑 Manually creating Signals requires remembering to call `.manually_drop()` on the signal whereas `use_signal` cleans the Signal up for you automatically.

Whenever a signal's value changes, its containing "reactive scope" will be "marked dirty" and re-run. By default, Dioxus components are reactive scopes, and thus, will re-render whenever a signal value changes.



Signals are core to Dioxus and take time to master. We recommend reading the [state management](#) guide in depth before diving into your first large app.

Global State with Context

While hooks are good for state *local* to components, occasionally you'll want to manage state for your *entire* app.

Dioxus provides two mechanisms: `Context` and `GlobalSignal`.

The `Context` API makes it possible for parent components to share state with child components without explicitly declaring an additional property field. This is used by larger apps and libraries to share state across the app without modifying component signatures.

To "provide" context, simply call `use_context_provider()` with a struct that implements `Clone`. To read the context in a child, call `use_context()`.

src/guide_state.rs

```
// Create a new wrapper type
#[derive(Clone)]
struct TitleState(String);

fn App() -> Element {
    // Provide that type as a Context
```

```

    use_context_provider(|| TitleState("HotDog".to_string()));
    rsx! {
        Title {}
    }
}

fn Title() -> Element {
    // Consume that type as a Context
    let title = use_context::<TitleState>();
    rsx! {
        h1 { "{title.0}" }
    }
}

```

You can combine `use_signal` and `Context` to provide reactive state to your app:

src/guide_state.rs

```

#[derive(Clone, Copy)]
struct MusicPlayer {
    song: Signal<String>,
}

fn use_music_player_provider() {
    let song = use_signal(|| "Drift Away".to_string());
    use_context_provider(|| MusicPlayer { song });
}

```

With `use_context` and `consume_context`, you can easily reach up to modify that state:

src/guide_state.rs

```

#[component]
fn Player() -> Element {
    rsx! {
        button {
            onclick: move |_|
consume_context::<MusicPlayer>().song.set("Vienna".to_string()),
            "Shuffle"
        }
    }
}

```

Any components that read the song signal will automatically re-render when the value changes.

Global Signals

Occasionally you'll want a simple global value. This is where `GlobalSignal` helps.

`GlobalSignals` are a combination of the `Context` system and `Signals` that require no additional structs or setup.

Simply declare a `GlobalSignal` somewhere in your app:

src/guide_state.rs

```
static SONG: GlobalSignal<String> = Signal::global(|| "Drift Away".to_string());
```

And then read and write to it from anywhere:

src/guide_state.rs

```
#[component]
fn Player() -> Element {
    rsx! {
        h3 { "Now playing {SONG}" }
        button {
            onclick: move |_| *SONG.write() = "Vienna".to_string(),
            "Shuffle"
        }
    }
}
```

🔗 GlobalSignals are only global to one app - not the entire program. On the server, every app gets its own GlobalSignal.

We won't need either GlobalSignal or Context for *HotDog*, but it's important to know that these are available to you.

Fetching Data

Our *HotDog* app has some basic interactivity but does not yet fetch new dog images. In this chapter, we'll interact with async and fetching data from an API.

Adding Dependencies

Dioxus does not provide any built-in utilities for fetching data. Crates like [dioxus-query](#) exist, but for this tutorial we'll implement data-fetching from scratch.

First, we need to add two new dependencies to our app: [serde](#) and [reqwest](#).

- Reqwest provides an HTTP client for fetching.
- Serde will let us derive a JSON Deserializer to decode the response.

In a new terminal window, add these crates to your app with `cargo add`.

```
cargo add reqwest --features json
cargo add serde --features derive
```

Defining a Response Type

We'll be using the amazing [dog.ceo/dog-api](#) to fetch images of dogs for *HotDog*. Fortunately, the API response is quite simple to deserialize.

Let's create a new Rust struct that matches the format of the API and derive `Deserialize` for it.

The Dog API docs outline a sample API response:

```
{
  "message": "https://images.dog.ceo/breeds/leonberg/n02111129_974.jpg",
  "status": "success"
}
```

Our Rust struct needs to match that format, though for now we'll only include the "message" field.

src/guide_data_fetching.rs

```
#[derive(serde::Deserialize)]
struct DogApi {
    message: String,
}
```

Using request and async

Dioxus has stellar support for asynchronous Rust. We can simply convert our `onclick` handler to be `async` and then set the `img_src` after the future has resolved.

The changes to our code are quite simple - just add the `request::get` call and then call `.set()` on `img_src` with the result.

src/guide_data_fetching.rs

```
#[component]
fn DogView() -> Element {
    let mut img_src = use_signal(|| "".to_string());

    let fetch_new = move |_| async move {
        let response = request::get("https://dog.ceo/api/breeds/image/random")
            .await
            .unwrap()
            .json::<DogApi>()
            .await
            .unwrap();

        img_src.set(response.message);
    };

    // ..

    rsx! {
        div { id: "dogview",
            img { src: "{img_src}" }
        }
        div { id: "buttons",
            // ..
            button { onclick: fetch_new, id: "save", "save!" }
        }
    }
}
```

Dioxus automatically calls `spawn` on asynchronous closures. You can also use `spawn` to perform async work *without* async closures - just call `spawn()` on any async block.

src/guide_data_fetching.rs

```
rsx! {
    button {
        onclick: move |_| {
            spawn(async move {
                // do some async work...
            });
        }
    }
}
```

The futures passed to `spawn` must not contain latent references to data outside the `async` block. Data that is *Copy* *can* be captured by `async` blocks, but all other data must be *moved*, usually by calling `.clone()`.

Managing Data Fetching with `use_resource`

Eventually, using bare `async` calls might lead to race conditions and weird state bugs. For example, if the user clicks the *fetch* button too quickly, then two requests will be made in parallel. If the request is updating data somewhere else, the wrong request might finish early and causes a race condition.

In Dioxus, *Resources* are pieces of state whose value is dependent on the completion of some asynchronous work. The `use_resource` hook provides a `Resource` object with helpful methods to start, stop, pause, and modify the asynchronous state.

Let's change our component to use a resource instead:

src/guide_data_fetching.rs

```
#[component]
fn DogView() -> Element {
    let mut img_src = use_resource(|| async move {
        request::get("https://dog.ceo/api/breeds/image/random")
            .await
            .unwrap()
            .json::<DogApi>()
            .await
            .unwrap()
            .message
    });

    rsx! {
        div { id: "dogview",
            img { src: img_src.cloned().unwrap_or_default() }
        }
        div { id: "buttons",
            button { onclick: move |_| img_src.restart(), id: "skip", "skip" }
            button { onclick: move |_| img_src.restart(), id: "save", "save!" }
        }
    }
}
```

Resources are very powerful: they integrate with `Suspense`, `Streaming HTML`, `reactivity`, and more.

The details of the `Resource` API are not terribly important right now, but you'll be using `Resources` frequently in larger apps, so it's a good idea to [read the docs](#).

Adding a Backend

Dioxus is a *fullstack* framework, meaning it allows you to seamlessly build your frontend alongside your backend.

We provide a number of utilities like *Server Functions*, *Server Futures*, and *Server State* for you to integrate into your apps. In this chapter, we'll cover loading and saving state to our backend with *Server Functions*. For an in-depth guide on fullstack, check out the dedicated [Fullstack Guide](#).


```
}    Ok(())
```

Every server function is an async function that takes some parameters and returns a `Result<(), ServerFnError>`. Whenever the client calls the server function, it sends an HTTP request to a corresponding endpoint on the server. The parameters of the server function are serialized as the body of the HTTP request. As a result, each argument must be serializable.

On the client, the server function expands to a `request` call:

src/guide_backend.rs

```
// on the client:
async fn save_dog(image: String) -> Result<(), ServerFnError> {
    request::Client::new()
        .post("http://localhost:8080/api/save_dog")
        .json(&image)
        .send()
        .await?;
    Ok(())
}
```

On the server, the server function expands to an [axum](#) handler:

src/guide_backend.rs

```
// on the server:
struct SaveDogArgs {
    image: String,
}

async fn save_dog(Json(args): Json<SaveDogArgs>) -> Result<(), ServerFnError> {
    Ok(())
}
```

When `dioxus::launch` is called, the server functions are automatically registered for you and set up as an Axum router.

src/guide_backend.rs

```
async fn launch(config: ServeConfig, app: fn() -> Element) {
    // register server functions
    let router = axum::Router::new().serve_dioxus_application(config, app);

    // start server
    let socket_addr = dioxus_cli_config::fullstack_address_or_localhost();
    let listener = tokio::net::TcpListener::bind(socket_addr).await.unwrap();
    axum::serve(listener, router).await.unwrap();
}
```

As of Dioxus 0.6, we only support the `axum` server framework. We plan to build additional server features in the future and only support `axum` to ship faster.

In some cases, the `dioxus::launch` function might be too limiting for your use-case on the server. You can easily drop down to `axum` by changing your `main.rs`. The `dioxus::launch` function also handles setting up logging and reading environment variables, which you will have to handle yourself.

src/guide_backend.rs

```
fn main() {
    #[cfg(feature = "server")]
    tokio::runtime::Runtime::new()
        .unwrap()
        .block_on(launch_server());
    #[cfg(not(feature = "server"))]
    dioxus::launch(App);
}

#[cfg(feature = "server")]
async fn launch_server() {
    // Connect to dioxus' logging infrastructure
    dioxus::logger::initialize_default();

    // Connect to the IP and PORT env vars passed by the Dioxus CLI (or your
    // dockerfile)
    let socket_addr = dioxus_cli_config::fullstack_address_or_localhost();

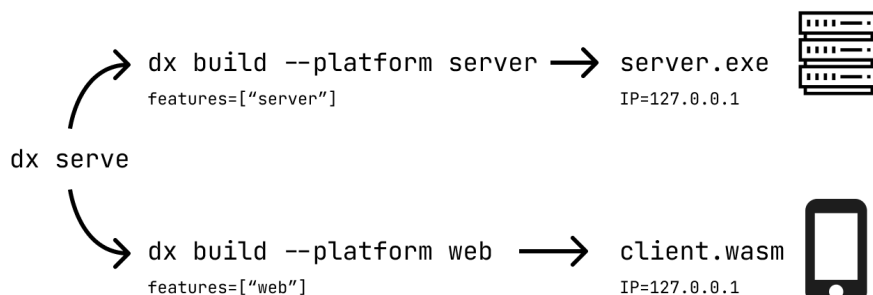
    // Build a custom axum router
    let router = axum::Router::new()
        .serve_dioxus_application(ServeConfigBuilder::new(), App)
        .into_make_service();

    // And launch it!
    let listener = tokio::net::TcpListener::bind(socket_addr).await.unwrap();
    axum::serve(listener, router).await.unwrap();
}
```

The Client/Server split

When Dioxus builds your fullstack apps, it actually creates two separate applications: the server and the client. To achieve this, dx passes different features to each build.

- The client is built with `--feature web`
- The server is built with `--feature server`



When embedding server code in our apps, we need to be careful about which code gets compiled. The body of the server function is designed to only be *executed on the server* - not the client. Any

code configured by the "server" feature will not be present in the final app. Conversely, any code not configured by the "server" feature *will* make it into the final app.

src/guide_backend.rs

```
// ✗ this will leak your DB_PASSWORD to your client app!
static DB_PASSWORD: &str = "1234";

#[server]
async fn DoThing() -> Result<(), ServerFnError> {
    connect_to_db(DB_PASSWORD).await
    // ...
}
```

Instead, we recommend placing server-only code within modules configured for the "server" feature.

src/guide_backend.rs

```
// ✓ code in this module can only be accessed on the server
#[cfg(feature = "server")]
mod server_utils {
    pub static DB_PASSWORD: &str = "1234";
}
```

In addition to the "server" feature, Dioxus expects a client side rendering feature like "web" or "desktop". Some libraries like web-sys only work when running in the browser, so make sure to not run specific client code in your server functions or before your `launch`. You can place client only code under a config for a client target feature like "web".

src/guide_backend.rs

```
fn main() {
    // ✗ attempting to use web_sys on the server will panic!
    let window = web_sys::window();

    // ..

    dioxus::launch(App);
}
```

Managing Dependencies

Some dependencies like [Tokio](#) only compile properly when targeting native platforms. Other dependencies like [jni-sys](#) only work properly when running on a specific platform. In these cases, you'll want to make sure that these dependencies are only compiled when a particular feature is enabled. To do this, we can use Rust's optional [flag on dependencies in our Cargo.toml](#).

```
[dependencies]
tokio = { version = "1", optional = true }

[features]
default = []
server = ["dep:tokio"]
```

Eventually, if your project becomes large enough, you might want to pull your server functions out into their own crate to be used across different apps. We'd create a `server` crate in our workspace:

```
└─ Cargo.toml
└─ crates
    └─ dashboard
    └─ marketplace
    └─ server
```

And then we'd import the server functions in our app, disabling their "server" feature.

```
[dependencies]
server = { workspace = true, default-features = false }
```

We provide a longer guide about the details of managing dependencies across the server and the client [here](#).

Our HotDog Server Function

Revisiting our HotDog app, let's create a new server function that saves our favorite dog to a file called `dogs.txt`. In production, you'd want to use a proper database as covered in [the next chapter](#), but for now we'll use a simple file to test things out.

src/guide_backend.rs

```
// Expose a `save_dog` endpoint on our server that takes an "image" parameter
#[server]
async fn save_dog(image: String) -> Result<(), ServerFnError> {
    use std::io::Write;

    // Open the `dogs.txt` file in append-only mode, creating it if it doesn't
    exist;
    let mut file = std::fs::OpenOptions::new()
        .write(true)
        .append(true)
        .create(true)
        .open("dogs.txt")
        .unwrap();

    // And then write a newline to it with the image url
    file.write_fmt(format_args!("{image}\n"));

    Ok(())
}
```

Calling the server function

Now, in our client code, we can actually call the server function.

src/guide_backend.rs

```
fn DogView() -> Element {
    let mut img_src = use_resource(snipped!());

    // ...
    rsx! {
        // ...
        div { id: "buttons",
            // ...
            button {
                id: "save",
                onclick: move |_| async move {
                    let current = img_src.cloned().unwrap();
```

```

        img_src.restart();
        _ = save_dog(current).await;
    },

    "save!"

}

}

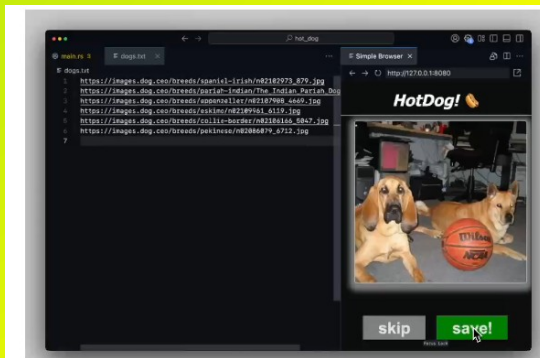
}

}

}

```

Wow, our app is really coming together!



Server functions are extremely capable and can even be used during server-side-rendering. Check out the complete [fullstack guide](#) for more information.

Working with Databases

Our HotDog app is coming together nicely! We implemented a very simple backend that saves the user's favorite dog images to a local "dogs.txt" file.

In practice, you will likely want to store data in a proper database. Modern databases are *much* more powerful than a text file!

If you already have a good understanding of databases, jump ahead to the [section where we integrate Sqlite with HotDog](#);

Choosing a Database

In today's age of app development, there are *many* databases to choose from, each with their own strengths, weaknesses, and tradeoffs to consider. For apps with just a few users, it is fine to select a "simpler" database that's easier to manage. For apps with many users, you might want to consider more advanced databases with additional tooling to meet stricter requirements.

Here is a (incomplete!) list of databases and a short summary of each:

- PostgreSQL: Advanced database known for its powerful plugin system.
- MySQL: World's most popular open source database good for all apps.
- SQLite: Simple file-based engine known for its reliability and embeddability.
- Oracle: Advanced commercial database known for its enterprise features.
- Redis: Simple key-value database known for its great performance.
- MongoDB: A database ideal for data that doesn't fit into rows and columns.

- [SurrealDB](#): A new "all-in-one" database that combines many models.
- [CockroachDB](#): Distributed SQL database designed for high-availability.
- [and many more!](#)

There are many different types of databases, each good at different tasks. These might include:

- **Relational**: traditional row/column/table approach.
- **Document**: storing unstructured or loosely structured blobs of data.
- **Timeseries**: storing and querying lots of data that changes over time.
- **Graph**: querying data based on its connections to other data.
- **Key-value**: storing just key-value pairs - a fast concurrent HashMap.
- **In-memory**: designed for low-latency operations usually used as a cache.
- **Embedded**: a database that is shipped *inside* your app.

For most apps - unless you have specific requirements - we recommend a mainstream relational database like PostgreSQL or MySQL.

 PostgreSQL is currently a very interesting option: it can be extended to support time-series, vector, graph, search and geo-spatial data with plugins.

In some cases, you might want a database that's specific to *just one app instance* or the *user's machine*. In these cases, you'll want to use an embedded database like [SQLite](#) or [RocksDB](#).

[Adding Database operations to HotDog](#)

For *HotDog*, we're going to use Sqlite. *HotDog* is a very simple app and will only ever have one user: you!

To add sqlite functionality to *HotDog*, we'll pull in the `rusqlite` crate. Note that `rusqlite` is only meant to be compiled on the server, so we'll feature gate it behind the "server" feature in our Cargo.toml.

```
[dependencies]
# ....
rusqlite = { version = "0.32.1", optional = true } # <--- add rusqlite
```

```
[features]
# ....
server = ["dioxus/server", "dep:rusqlite"] # <---- add dep:rusqlite
```

To connect to our database, we're going to use the `rusqlite::Connection`. Rusqlite connections are not thread-safe and must exist once-per-thread, so we'll need to wrap it in a `thread_locals`.

When the connection is initialized, we'll run a SQL action to create the "dogs" table with our data.

src/guide_databases.rs

```
// The database is only available to server code
#[cfg(feature = "server")]
thread_local! {
    pub static DB: rusqlite::Connection = {
        // Open the database from the persisted "hotdog.db" file
```

```

let conn = rusqlite::Connection::open("hotdog.db").expect("Failed to
open database");

// Create the "dogs" table if it doesn't already exist
conn.execute_batch(
    "CREATE TABLE IF NOT EXISTS dogs (
        id INTEGER PRIMARY KEY,
        url TEXT NOT NULL
    );",
).unwrap();

// Return the connection
conn
};
}

```

Now, in our `save_dog` server function, we can use SQL to insert the value into the database:

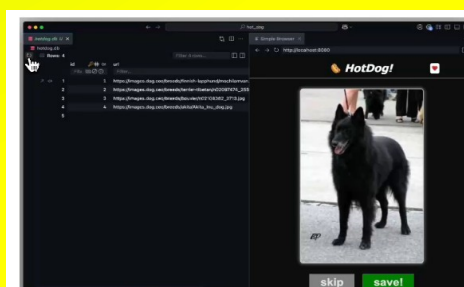
`src/guide_databases.rs`

```

#[server]
async fn save_dog(image: String) -> Result<(), ServerFnError> {
    DB.with(|f| f.execute("INSERT INTO dogs (url) VALUES (?1)", &[&image]))?;
    Ok(())
}

```

Once the app is launched, you should see a "hotdog.db" file in your crate's directory. Let's save a few dog photos and then open the database in a database viewer. If all goes well, you should see the saved dog photos!



Notes on Databases and Rust

While there are many database providers, Rust support can be limited. Rust is still a new choice for web development. In this section we'll provide our own (biased!) opinions on what libraries we recommend for interacting with databases.

It's also important to note that several libraries exist at a higher level abstraction than raw SQL. These are called an *Object Relationship Mapper (ORM)*. Rust ORM libraries map the SQL language into ordinary Rust functions. We generally recommend just sticking with SQL, but ORMs can make working writing some queries easier.

- [Sqlx](#): A straightforward yet large interface to Postgres, MySql, and Sqlite.
- [SeaORM](#): An ORM built on top of Sqlx for deriving databases.
- [rusqlite](#): An intuitive sqlite interface with no special ORM magic.
- [rust-postgres](#): An interface to Postgres with an API similar to rusqlite.
- [Turbosql](#): A very terse interface to Sqlite with automatic derives.

We aren't including libraries like [Diesel](#) in this list since it seems that the Rust ecosystem has evolved towards newer projects with 1st-class async support.

There are many libraries we haven't tested yet, but might be worth checking out:

- [firebase-rs](#): Firebase client crate
- [postgrest-rs](#): Supabase client crate
- [mongo-rust-driver](#): Official MongoDB client crate

Choosing a Database Provider

While there are just a handful of databases you might consider for your app, there are many *database providers*, each with their own strengths and weaknesses. We are not sponsored by any of these providers - this is just a list of providers we have seen in use by Rust apps.

You *do not* need to use a database provider. Databases providers provide paid database hosting. It will cost you money to use these providers! Many have a free tier and some support "scale-to-zero" to help you save money on small apps. At any time, you are free to host and manage your own database.

For popular relational databases:

- [GCP](#): Provides AlloyDB (enterprise postgres), CloudSQL (MySQL, Postgres), and more.
- [AWS](#): Provides RDS, Aurora, DynamoDB, and more.
- [PlanetScale](#): Reliable MySQL-compatible database with sharding designed for scale.
- [Firebase](#): Google's comprehensive real-time database designed for rapid app development.
- [Supabase](#): Hosted Postgres known for its great dashboard and tooling.
- [Neon](#): Hosted Postgres that separates compute and storage for scale-to-zero apps.

For Sqlite:

- [LiteFS](#): A distributed Sqlite sync engine designed to be used with Fly.io
- [Turso](#): A "multi-tenant" sqlite provider that maintains one isolated database per user

The "scale-to-zero" relational solutions:

- [AWS Aurora](#)
- [LiteFS](#)

We don't suggest any particular database provider.

- If you have lots of free cloud credits, consider AWS/GCP/Azure.
- If you want Postgres with a good dashboard, consider Supabase or Neon.
- If you want a simple experience, consider Turso or LiteFS.

Adding More Routes

So far, our app has only had a single page. Let's change that!

In this chapter, we'll be adding a Navbar, a welcome screen, and a "favorites" page where we can revisit our favorite dogs.

Organizing our Project

Before we get too far with adding new pages to our app, let's organize our codebase a bit better. For larger projects you might want to break your app into different smaller crates. For HotDog, we'll keep it simple.

The `dx new` Jumpstart and Workspace templates provide great scaffolding for new apps!

We generally recommend splitting your components, models, and backend functionality into different files. For HotDog, we're going to use a simple directory structure:



We'll have a `backend.rs` that contains our server functions and a `components` folder that contains our components. We don't have a `NavBar` or a `Favorites` component yet, but we'll still create the relevant files before adding them. By splitting out our server functions into a `backend.rs` file, we'll make it easier to extract our backend functionality as a shared library for different apps in the future.

Our `components/mod.rs` file will simply import and re-export the components in `view.rs`, `nav.rs`, and `favorites.rs`:

```
mod favorites;
mod nav;
mod view;

pub use favorites::*;
pub use nav::*;
pub use view::*;
```

Finally, we need to bring `backend` and `components` into scope in our `main.rs` file:

```
mod components;
mod backend;

use crate::components::*;
```

For more information on organizing Rust projects with modules, see the [Modules section](#) of the Rust Book.

Creating a Route

Most Dioxus apps you'll build will have different screens. This could include pages like *Login*, *Settings*, and *Profile*. Our HotDog app will have two screens: a *DogView* page and a *Favorites* page.

Dioxus provides a first-party router that natively integrates with web, desktop, and mobile. For example, on web, whenever you visit the `/favorites` url in your browser, the corresponding *Favorites* page will load. The Dioxus router is very powerful, and most importantly, type-safe. You can rest easy knowing that users will never be sent to an invalid route. To achieve this, we first need to add the "Router" feature to the Cargo.toml file:

```
[dependencies]
dioxus = { version = "0.6.0", features = ["fullstack", "router"] } # <----- add "router"
```

Next, the Dioxus router is defined as an enum with the `Routable` derive attribute:

src/guide_router.rs

```
#[derive(Routable, Clone, PartialEq)]
enum Route {
    #[route("/")]
    DogView,
}
```

With the Dioxus router, every route is an enum variant with a `#[route]` attribute that specifies the route's URL. Whenever the router renders our route, the component of the same name will be rendered.

src/guide_router.rs

```
use dioxus::prelude::*;

#[derive(Routable, Clone, PartialEq)]
enum Route {
    #[route("/")]
    DogView, // <----- a DogView component must be in scope
}

fn DogView() -> Element {
    todo!()
}
```

Rendering the Route

Now that we have our app's `Route` defined, we need to render it. Let's change our `app` component to render the `Route {}` component instead of the `DogView`.

src/guide_router.rs

```
fn app() -> Element {
    rsx! {
        document::Stylesheet { href: asset!("/assets/main.css") }

        // 🗑 delete Title and DogView and replace it with the Router component.
        Router::<Route> {}
    }
}
```

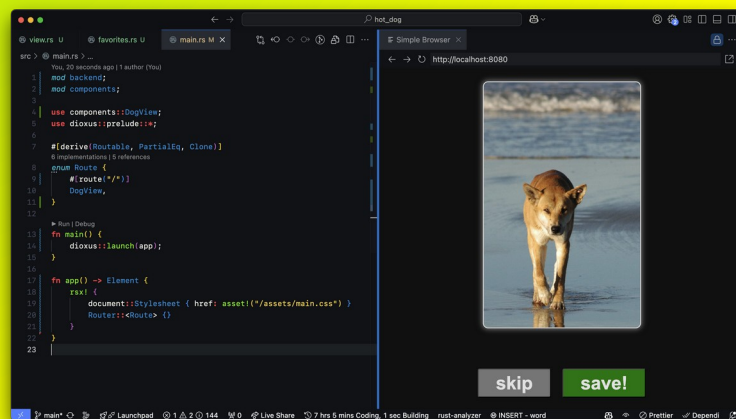
When the `Router {}` component renders, it will parse the document's current URL into a `Route` variant. If the url doesn't parse properly, the router will render nothing unless you add a "catch-all" route:

src/guide_router.rs

```
#[derive(Routable, Clone, PartialEq)]
enum Route {
    // ...
    // We can collect the segments of the URL into a Vec<String>
    #[route("/:..segments")]
    PageNotFound { segments: Vec<String> },
}
```

Note here that the `PageNotFound` route takes the "segments" parameter. Dioxus routes are not only type-safe as variants, but also type-safe with URL parameters. For more information on how this works, [check the router guide](#).

At this point, we should see our app, but this time without its Title.



Rendering the NavBar with a Layout

We're rendering our `DogView` component, but unfortunately we no longer see our Title. Let's add that back and turn it into a `NavBar`!

In our `src/components/nav.rs` file, we'll add back our Title code, but rename it to `NavBar` and modify it with two new items: the `Link {}` and `Outlet` components.

src/guide_router.rs

```
use crate::Route;
use dioxus::prelude::*;

#[component]
pub fn NavBar() -> Element {
    rsx! {
        div { id: "title",
            Link { to: Route::DogView,
                h1 { "🐕 HotDog! " }
            }
        }
        Outlet::<Route> {}
    }
}
```

The `Link {}` component wraps the anchor `<a>` element with a type-safe interface. This means any struct that implements `Routable` - anything that can `.to_string()` - is a valid navigation target.

src/guide_router.rs

```
// Using the Link with Route
Link { to: Route::DogView }

// Or passing in a "/" route directly
Link { to: "/" }
```

The `Link` component takes many different arguments, making it possible to extend and customize for your use-case.

In `NavBar`, we also added an `Outlet::<Route> {}` component. When the `Router` component renders, it first looks for any child `Outlet` components. If one is present, it renders the current route *under the outlet*. This lets us wrap the current page in extra elements - in this case, the `NavBar`. If no `Outlet` is present, then the current route is simply rendered where the `Router {}` is declared.

To actually add the `NavBar` component to our app, we need to update our `Route` enum with the `#[layout]` attribute. This forces the router to render the `NavBar` component *first* so it can expose its `Outlet {}`.

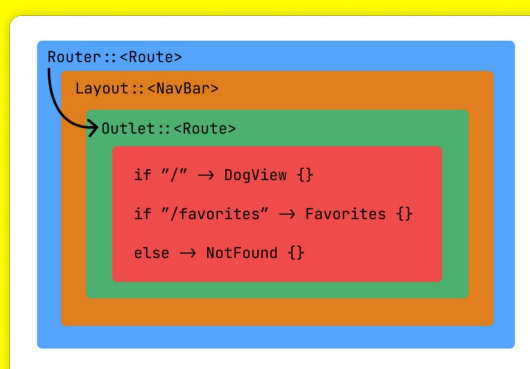
src/guide_router.rs

```
#[derive(Routable, PartialEq, Clone)]
enum Route {
    #[layout(NavBar)] // <---- add the #[layout] attribute
    #[route("/")]
    DogView,
}
```

The `layout` attribute instructs the `Router` to wrap the following enum variants in the given component.

```
Router {
    NavBar {
        Outlet {
            if route == "/" {
                DogView {}
            }
        }
    }
}
```

Visually, this should be straight-forward to understand. Note that the `Router` and `Outlet` share the same `Route` generic type.



Adding a Favorites Route

Now that we understand the fundamentals of routing, let's finally add our *Favorites* page so we can view the dog photos we saved.

We'll start by creating an empty component `src/components/favorites.rs`:

`src/guide_router.rs`

```
use dioxus::prelude::*;

#[component]
pub fn Favorites() -> Element {
    rsx! { "favorites!" }
}
```

And then let's make sure to add a new variant in our `Route` enum:

`src/guide_router.rs`

```
#[derive(Routable, PartialEq, Clone)]
enum Route {
    #[layout(NavBar)]
    #[route("/")]
    DogView,

    #[route("/favorites")]
    Favorites, // <----- add this new variant
}
```

To make sure the user can reach this page, let's also add a button in the nav that points to it.

`src/guide_router.rs`

```
use crate::Route;
use dioxus::prelude::*;

#[component]
pub fn NavBar() -> Element {
    rsx! {
        div { id: "title",
            Link { to: Route::DogView,
                h1 { "🌯 HotDog! " }
            }
            Link { to: Route::Favorites, id: "heart", "♥" } // <----- add this
        }
        Outlet::<Route> {}
    }
}
```

Our Favorites Page

Finally, we can build our favorites page. Let's add a new `list_dogs` server function that fetches the 10 most recently saved dog photos:

`src/guide_router.rs`

```
// Query the database and return the last 10 dogs and their url
#[server]
pub async fn list_dogs() -> Result<Vec<(usize, String)>, ServerFnError> {
    let dogs = DB.with(|f| {
        f.prepare("SELECT id, url FROM dogs ORDER BY id DESC LIMIT 10")
            .unwrap()
            .query_map([], |row| Ok((row.get(0)?, row.get(1)?)))
            .unwrap()
            .map(|r| r.unwrap())
            .collect()
    });

    Ok(dogs)
}
```

Now, we can fill in our component. We're going to use the same `use_resource` hook from earlier. Resolving the request from the server might take some time, so we'll use the `.suspend()` method on `Resource` to wait for the request to finish before mapping the contents to a list.

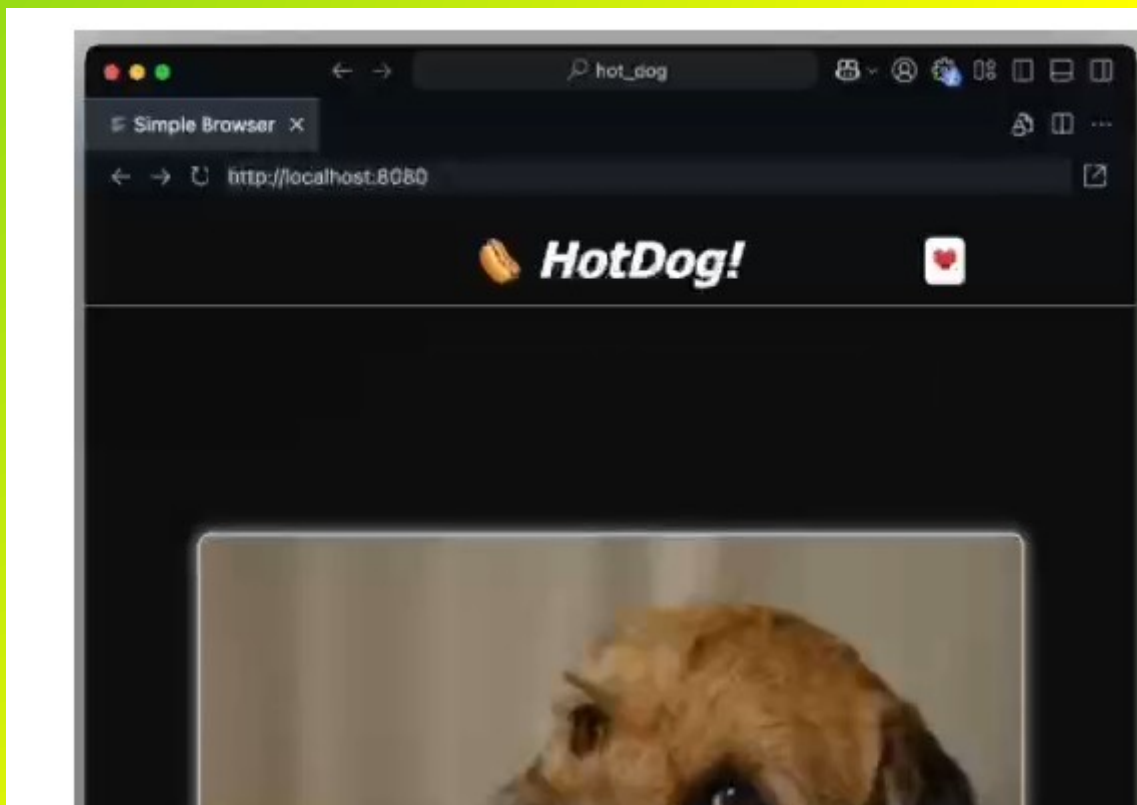
src/guide_router.rs

```
use dioxus::prelude::*;

#[component]
pub fn Favorites() -> Element {
    // Create a pending resource that resolves to the list of dogs from the
    // backend
    // Wait for the favorites list to resolve with `.suspend()`
    let mut favorites = use_resource(super::backend::list_dogs).suspend()?;

    rsx! {
        div { id: "favorites",
            div { id: "favorites-container",
                for (id, url) in favorites().unwrap() {
                    // Render a div for each photo using the dog's ID as the
                    list key
                    div {
                        key: id,
                        class: "favorite-dog",
                        img { src: "{url}" }
                    }
                }
            }
        }
    }
}
```

As a stretch goal, try adding a button that lets the user also delete items from the database.



Bundling

Congratulations! You built your first fully-functional Dioxus app, completely loaded with Routing, asynchronous data-fetching, Server Functions, and a database! That's incredible for just a few minutes of work.

Let's get your app bundled for multiple platforms and then ready to deploy.

Testing on Desktop and Mobile

So far, we've been testing our app in a simple web browser. Let's actually build and test our app for mobile platforms.

In Dioxus 0.6, dx finally supports `dx serve` for Android and iOS!

Testing on iOS

To test iOS, your development environment needs to be setup to build iOS apps. This involves a few steps:

- Make sure you are developing on a device running macOS
- Install XCode
- [Download a recent iOS SDK and Emulator pack](#)
- Install the iOS Rust toolchains (`aarch64-apple-ios` `aarch64-apple-ios-sim`)

This is a multi-step process and requires creating an Apple Developer account. You shouldn't need to pay any fees until you want to sign your app. Signing your app is required for deploying to the Apple App Store and testing on your iOS device.

If everything is installed properly, you should be able to open the Simulator app:

```
open /Applications/Xcode.app/Contents/Developer/Applications/Simulator.app
```

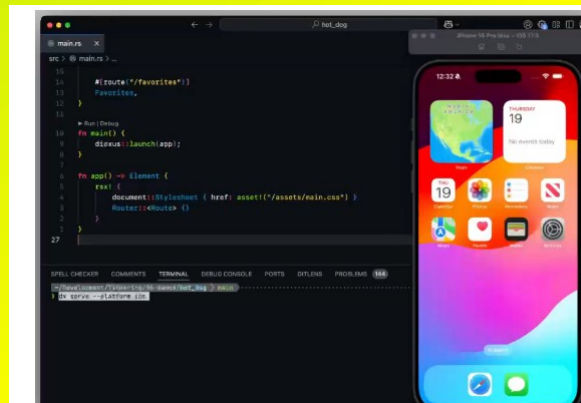
If the Simulator app opens but no device pops up, you might need to open a specific device. Use `xcrun` to discover which devices you have installed.

```
xcrun simctl list
```

Identify an available device. We're going to simulate an iPhone 15 Pro Max:

```
xcrun simctl boot "iPhone 15 Pro Max"
```

Once the simulator is booted, we can run `dx serve --platform ios`.



Fantastic - our app works seamlessly with no changes.

Testing on Android

Setting up your environment for Android development takes time, so make sure to read the [mobile tooling guide](#).

- Install the Android NDK and SDK
- Set `JAVA_HOME`, `ANDROID_HOME`, `NDK_HOME`, and fix `PATH` issues to use the emulator tool
- Install and set up an Android emulator
- Install the Android rustup targets (`aarch64-linux-android` `armv7-linux-androideabi` `i686-linux-android` `x86_64-linux-android`)

Let's start an emulator. We can use the `emulator` command which should be in your `PATH` if setup properly. We're going to use our `Pixel_6_API_34` emulator, but you can use any device you've configured.

```
emulator -avd Pixel_6_API_34 -netdelay none -netspeed full
```

If we try to `dx serve --platform android`, we'll find that our app fails to build for Android. This is not good!

```

12:45:39 [cargo] Could not find directory of OpenSSL installation, and this `
sys` crate cannot
12:45:39 [cargo] proceed without this knowledge. If OpenSSL is installed and
this crate had
12:45:39 [cargo] trouble finding it, you can set the `OPENSSL_DIR`
environment variable for the
12:45:39 [cargo] compilation process.
12:45:39 [cargo] Make sure you also have the development packages of openssl
installed.
12:45:39 [cargo] For example, `libssl-dev` on Ubuntu or `openssl-devel` on
Fedora.
12:45:39 [cargo] If you're in a situation where you think the directory
*should* be found
12:45:39 [cargo] automatically, please open a bug at
https://github.com/sfackler/rust-openssl
12:45:39 [cargo] and include information about your system as well as this
message.
12:45:39 [cargo] $HOST = aarch64-apple-darwin
12:45:39 [cargo] $TARGET = aarch64-linux-android
12:45:39 [cargo] openssl-sys = 0.9.104

```

Currently, `rust-openssl` does not cross-compile properly for Android targets. To fix this, we need to add the `openssl` crate to our `Cargo.toml` and then enable its "vendored" feature. This will build OpenSSL from source instead of trying and failing to read it from the Android NDK.

We're only going to enable the vendored feature when targeting Android.

```

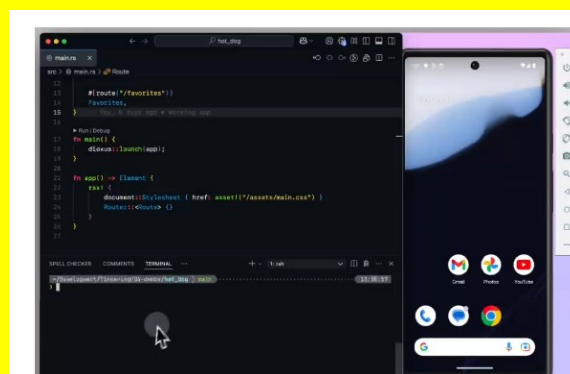
[target.'cfg(target_os = "android")'.dependencies]
openssl = { version = "0.10", features = ["vendored"]}

```

In the future, Dioxus might add OpenSSL's vendored feature implicitly to make this error go away. We're covering it here since it's important to understand that not every Rust dependency works out-of-the-box for iOS and Android. Unfortunately, the Rust ecosystem for mobile is still quite young and you'll need to know how to solve problems like these.

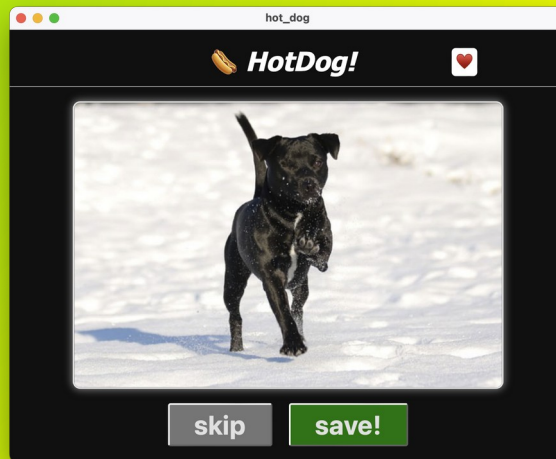
Let's try again!

```
dx serve --platform android
```



Testing on Desktop

HotDog also works on macOS, Windows, and Linux! We can use `dx serve --platform desktop` to serve our app as a desktop app.



Bundling for the web

After we're done making changes to our server and client apps, we can build bundles that are ready to distribute.

We're going to follow the same pattern as `dx serve` but with `dx bundle`. To start, let's build the web version of our app.

```
dx bundle --platform web
```

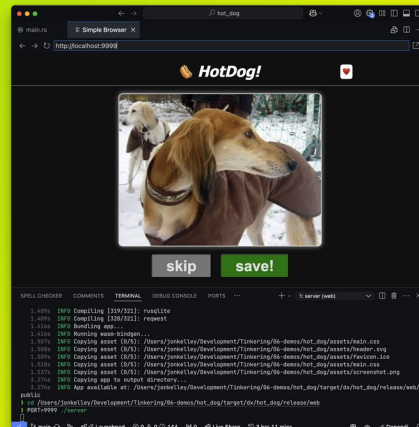
We should receive a series of INFO traces from the CLI as it builds, and then finally a path to the `public` folder it generates. Let's `cd` into its public directory and then check out its parent directory (`cd ..`) (the "web" folder).

```
> tree -L 3 --gitignore
.
├── public
│   ├── assets
│   │   ├── favicon.ico
│   │   ├── header.svg
│   │   ├── main-14aa55e73f669f3e.css
│   │   ├── main.css
│   │   └── screenshot.png
│   ├── index.html
│   └── wasm
│       ├── hot_dog.js
│       ├── hot_dog.js.br
│       ├── hot_dog_bg.wasm
│       ├── hot_dog_bg.wasm.br
│       └── snippets
└── server
```

`dx` built a `public` folder containing our assets, `index.html`, and various JavaScript snippets. Alongside our `public` folder is a `server` binary. When we deploy our web assets, we'll also want to deploy the server since it provides our server functions.

We can manually run the server simply by executing it. If you're using a default `dioxus::launch` setup, then the server will read the `IP` and `PORT` environment variables to serve.

🔧 If you intend to serve from within a container (e.g., Docker), then you need to override the default `127.0.0.1` address with `IP=0.0.0.0` to listen for external connections.



Bundling for Desktop and Mobile

To bundle desktop and mobile apps for deployment, we'll again use `dx bundle`. As of today, `dx bundle` only builds desktop apps for the native platform and architecture. Unfortunately, you can't build macOS apps from Windows, Linux apps from Mac, etc. We recommend using a Continuous Integration Matrix (like [Github Actions](#)) to perform a "cross-build" of your app in multiple different containers.

When bundling installable apps, there are many distribution formats to choose from. We can specify these formats using the `--package-types` flag on `dx bundle`. Dioxus supports packaging a broad number of package types:

- macOS: `.app`, `.dmg`
- Linux: `.appimage`, `.rpm`, `.deb`
- Windows: `.msi`, `.exe`
- iOS: `.app`
- Android: `.apk`

You can specify package types like so:

```
dx bundle --platform desktop \
  --package-types "macos" \
  --package-types "dmg"
```

Note that not all package-types are compatible with each platform - eg. only `.exe` can be built when specifying `--platform desktop`.

We should see the outputs in our terminal:

```
18.252s INFO Bundled app successfully!
18.252s INFO App produced 2 outputs:
18.252s INFO app - [/Users/jonkelley/Development/Tinkering/06-demos/hot_dog/target/dx/hot_dog/bundle/macos/bundle/macos/HotDog.app]
```

```
18.252s INFO dmg - [/Users/jonkelley/Development/Tinkering/06-demos/hot_dog/target/dx/hot_dog/bundle/macos/bundle/dmg/HotDog_0.1.0_aarch64.dmg]
```

Generally, you can distribute desktop apps without needing an app store. However, some platforms like macOS might require you to sign and notarize your application to be considered "safe" for your users to open.

When distributing mobile apps, you *are required* to sign and notarize your apps. Currently, Dioxus doesn't provide built-in utilities for this, so you'll need to figure out signing by reading 3rd-party documentation.

Tauri provides documentation on the signing process:

- [macOS](#)
- [iOS](#)
- [Android](#)
- [Windows](#)
- [Linux](#)

Customizing your Bundle

Before you ship your app, you might want to configure how your app icon looks, what entitlements it has, and other details. Our `dx bundle` tool can help you configure your bundles in a variety of ways.

To configure our bundle, we'll use our `Dioxus.toml` and modify the bundle section.

```
[application]
name = "docsite"

[bundle]
identifier = "com.dioxuslabs"
publisher = "DioxusLabs"
icon = ["assets/icon.png"]
```

For a full list of options, see the [reference page on the bundle section](#).

Automating dx bundle with JSON mode

Also added in Dioxus 0.6 is a JSON output mode for `dx`. This makes it possible to parse the output of the CLI using tools like `jq` which provide stdin/stdout support for JSON parsing.

This mode is not particular friendly to humans, but does contain more information than the standard trace output.

```
{"timestamp":" 9.927s","level":"INFO","message":"Bundled app successfully!","target":"dx::cli::bundle"}
{"timestamp":" 9.927s","level":"INFO","message":"App produced 2 outputs:","target":"dx::cli::bundle"}
{"timestamp":" 9.927s","level":"DEBUG","message":"Bundling produced bundles:
[\n  Bundle {\n    package_type: MacOSBundle,\n    bundle_paths: [\n      \"/Users/jonkelley/Development/Tinkering/06-demos/hot_dog/target/dx/hot_dog/bundle/macos/bundle/macos/HotDog.app",\n    ],\n  },\n  Bundle {\n    package_type: Dmg,\n    bundle_paths: [\n
```

```

n
"/Users/jonkelley/Development/Tinkering/06-demos/hot_dog/target/dx/hot_dog/
bundle/macos/bundle/dmg/HotDog_0.1.0_aarch64.dmg",\n      ],\n    },\n
n]","target":"dx::cli::bundle"}
{"timestamp":" 9.927s","level":"INFO","message":"app -
[/Users/jonkelley/Development/Tinkering/06-demos/hot_dog/target/dx/hot_dog/
bundle/macos/bundle/macos/HotDog.app]","target":"dx::cli::bundle"}
{"timestamp":" 9.927s","level":"INFO","message":"dmg -
[/Users/jonkelley/Development/Tinkering/06-demos/hot_dog/target/dx/hot_dog/
bundle/macos/bundle/dmg/HotDog_0.1.0_aarch64.dmg]","target":"dx::cli::bundle"}
{"timestamp":" 9.927s","level":"DEBUG","json":{"\"BundleOutput\":{\"bundles\":
[\"/Users/jonkelley/Development/Tinkering/06-demos/hot_dog/target/dx/hot_dog/
bundle/macos/bundle/macos/HotDog.app\", \"/Users/jonkelley/Development/
Tinkering/06-demos/hot_dog/target/dx/hot_dog/bundle/macos/bundle/dmg/
HotDog_0.1.0_aarch64.dmg\"]}}","target":"dx"}

```

JSON mode works with all `dx` commands. However, it is most useful with `dx build` and `dx bundle`. The CLI always guarantees that the last emitted line is the result of the command. To collect the list of bundles from the `dx bundle` command, we can use `tail -1` and simple `jq`.

```

dx bundle --platform desktop \
  --json-output \
  --verbose \
  | tail -1 \
  | jq -r '.json | fromjson | .BundleOutput.bundles []'

```

This returns the list of bundles:

```

/Users/jonkelley/Development/Tinkering/06-demos/hot_dog/target/dx/hot_dog/
bundle/macos/bundle/macos/HotDog.app
/Users/jonkelley/Development/Tinkering/06-demos/hot_dog/target/dx/hot_dog/
bundle/macos/bundle/dmg/HotDog_0.1.0_aarch64.dmg

```

Deploying

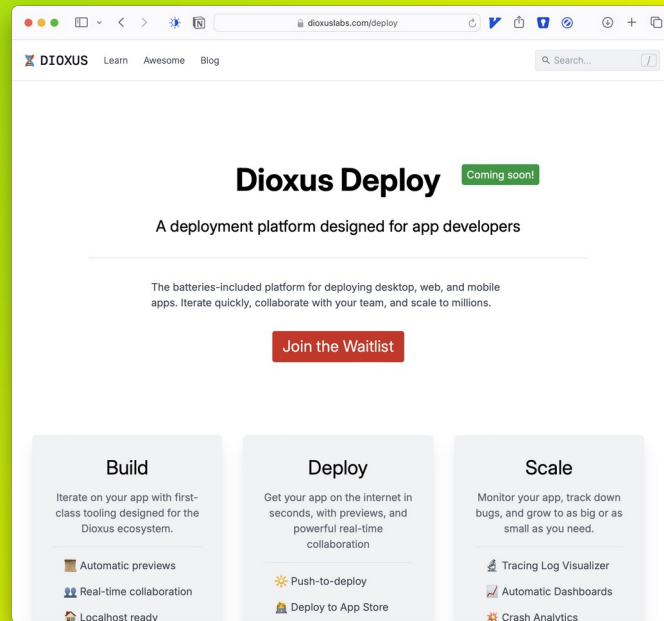
We're *finally* ready to deploy our bundled apps into the world. Congrats on making it this far!

This step is optional for the tutorial but worth covering to understand the process. Feel free to skip ahead to [next steps](#) if you're not interested in deploying.

Dioxus Deploy

As mentioned in the [introduction](#), Dioxus is an independent project with aspirations to fund itself through a paid deploy platform. Hopefully, one day, enough people ship apps with [Dioxus Deploy](#) to fund development on Dioxus itself!

Currently, Dioxus does not provide its own deploy platform. If you want to sign-up for the beta and help us design the ideal "end-to-end app-development experience," please [join the waitlist!](#)



Deploying your Desktop and Mobile apps

Generally, deploying a desktop app is as simple as distributing the bundle directly. Simply upload your app bundles to a host like GitHub or S3. With a download link, your users can easily download and install your apps.

✏ When shipping fullstack apps to production, you'll want to make sure to set your backend API URL properly as [covered later](#).

If you'd like to distribute your app through app stores, you'll need to follow some additional steps.

- [iOS](#): Directly publish to the Apple App Store
- [macOS](#): Directly publish to the Apple App Store
- [Android](#): Directly publish to the Google Play Store

Tauri provides some [helpful guides](#) for deploying Tauri apps which, while not Dioxus apps, need to follow many of the same steps for deploying to app stores.

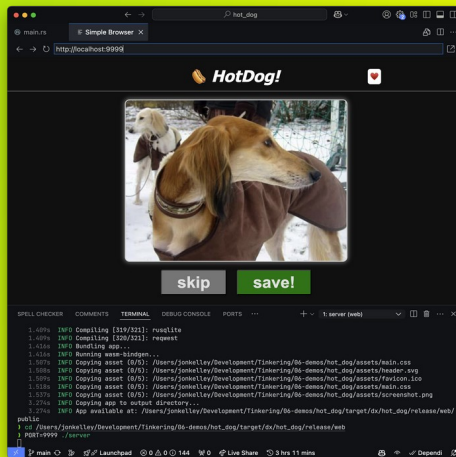
Making native app distribution easier is a top priority for Dioxus Deploy!

Deploy Requirements

Dioxus web apps are structured as a Client bundle and a Server executable. Generally, any deploy provider that exposes a simple container will be sufficient for a Dioxus fullstack web application.

Some providers like [Cloudflare Workers](#) and [Fermion Spin](#) provide WASM-based containers for apps. WASM runtimes are typically cheaper to operate and can horizontally scale better than a traditional virtual-machine based container. When deploying on WASM runtimes, you will need to create a WASM build of your server manually.

Running the webserver is as simple as executing `./server`. Make sure to set the IP and PORT environment variables correctly:



Choosing a deploy provider

There are *many* deploy providers! We're not going to get too deep into the pros/cons of any particular provider. Generally, providers are good at one of a few categories: price, performance, UI/UX, advanced features, and enterprise requirements.

Depending on your app, you might have strict requirements like SOC2 or HIPAA compliance. Make sure to do your own research for your own use-case.

- [AWS](#): Full-featured cloud provider powered by Amazon.
- [GCP](#): Full-featured cloud provider powered by Google.
- [Azure](#): Full-featured cloud provider powered by Microsoft.
- [Fly.io](#): Simple scale-to-zero micro-vm-based cloud with integrated wireguard.
- [Vercel](#): Developer-focused cloud built on AWS cloud functions popular with JavaScript frameworks.
- [Render](#): A "Modern Heroku" focused on developer experience and simplicity.
- [Digital Ocean](#): A cloud built around virtual machines, databases, and storage.

For *HotDog* we're going to deploy on [Fly.io](#). We like [Fly.io](#) for a number of reasons. Most importantly, Fly is built on Amazon's [Firecracker](#) project which is entirely written in Rust!

Fly is also quite simple to get started - just log in with either your GitHub account or Google account.

Building a Dockerfile

Some deploy providers have prebuilt solutions for various runtimes. For example, some have dedicated NodeJS and Python runtimes with strict requirements.

With Rust apps, there generally isn't a prebuilt "pack" to target. In these cases, we need to write a simple Dockerfile which compiles and starts our apps.

Our Dockerfile will have three phases. The first phase downloads and caches dependencies so incremental builds stay fast:

```
FROM rust:1 AS chef
RUN cargo install cargo-chef
WORKDIR /app
```

```
FROM chef AS planner
COPY . .
RUN cargo chef prepare --recipe-path recipe.json
```

In the second phase, we use cargo chef to load cached dependencies and preform the build:

```
FROM chef AS builder
COPY --from=planner /app/recipe.json recipe.json
RUN cargo chef cook --release --recipe-path recipe.json
COPY . .

# Install `dx`
RUN curl -L --proto '=https' --tlsv1.2 -sSf
https://raw.githubusercontent.com/cargo-bins/cargo-binstall/main/install-from-
binstall-release.sh | bash
RUN cargo binstall dioxus-cli --root /.cargo -y --force
ENV PATH="/.cargo/bin:$PATH"

# Create the final bundle folder. Bundle always executes in release mode with
optimizations enabled
RUN dx bundle --platform web
```

Finally, we copy the built "web" folder to the "slim" runtime that serves our app.

```
FROM chef AS runtime
COPY --from=builder /app/target/dx/hot_dog/release/web/ /usr/local/app

# set our port and make sure to listen for all connections
ENV PORT=8080
ENV IP=0.0.0.0

# expose the port 8080
EXPOSE 8080

WORKDIR /usr/local/app
ENTRYPOINT [ "/usr/local/app/server" ]
```

It's also a smart idea to set up a .dockerignore file:

```
**/target
**/dist
LICENSES
LICENSE
temp
README.md
```

Deploying to Fly

To get started with Fly, we need to go through the [Sign Up flow](#) and enter our details. This shouldn't take too long.

We'll add the dockerfile from above along with the dockerignore. We'll want to [install flyctl](#) which also installs the fly CLI.

Let's call [fly launch](#) which will automatically initialize our fly.toml.

```

> fly launch
Scanning source code
Detected a Dockerfile app
Creating app in /Users/jonkelley/Development/Tinkering/06-demos/hot_dog
We're about to launch your app on Fly.io. Here's what you're getting:

Organization: Jonathan Kelley      (fly launch defaults to the personal org)
Name:         hot-dog              (derived from your directory name)
Region:       San Jose, California (US) (this is the fastest region for you)
App Machines: shared-cpu-1x, 1GB RAM (most apps need about 1GB of RAM)
Postgres:     <none>               (not requested)
Redis:        <none>               (not requested)
Tigris:       <none>               (not requested)

? Do you want to tweak these settings before proceeding? (y/N)

```

`fly launch` will spin up a build machine for us and build our app. In a minute or two, our app should be fully built and deployed.

If we ever want to re-deploy our code, we can run `fly deploy`.

```

$ fly deploy
--> Verifying app config
Validating /Users/jonkelley/Development/Tinkering/06-demos/hot_dog/fly.toml
- Configuration is valid
--> Verified app config
--> Building image
Waiting for depot builder...

```

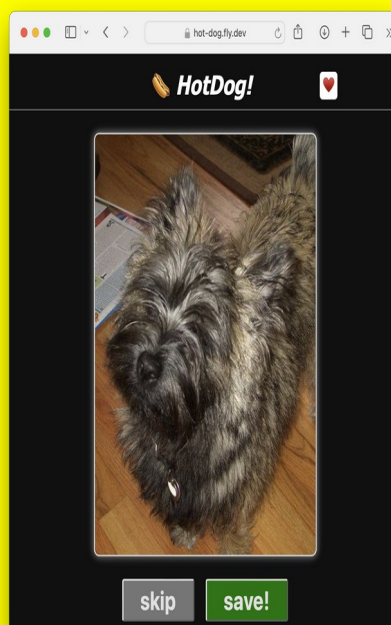
We can also add a volume to our app to persist our Sqlite database by adding a `[mounts]` section to our `Fly.toml`:

```

[mounts]
source = "hotdogdb"
destination = "/usr/local/app/hotdogdb"

```

Once the build is complete, Fly will assign our app a URL that we can customize later. With any luck, our app should be live!



Continuous Deployment

Fly also supports [continuous deployment](#). Whenever we push to our GitHub repository, we can execute `fly deploy` automatically. This can serve as a foundation for staging environments and automatic releases.

Our app just needs a `.github/workflows/fly-deploy.yml`.

```
name: Fly Deploy
on:
  push:
    branches:
      - main
jobs:
  deploy:
    name: Deploy app
    runs-on: ubuntu-latest
    concurrency: deploy-group
    steps:
      - uses: actions/checkout@v4
      - uses: superfly/flyctl-actions/setup-flyctl@master
      - run: flyctl deploy --remote-only
    env:
      FLY_API_TOKEN: ${ secrets.FLY_API_TOKEN }
```

Fullstack Desktop and Mobile

Now that our backend is live, we can wire up the API to our native apps. By default, Dioxus doesn't know where to find your API, so you'll need to specify the URL manually by calling `server_fn::client::set_server_url`.

src/guide_deploy.rs

```
fn main() {
  #[cfg(not(feature = "server"))]
  server_fn::client::set_server_url("https://hot-dog.fly.dev");

  dioxus::launch(App);
}
```

Note that as our app changes, the "true" endpoint of our server functions might change. The `#[server]` macro generates an API endpoint with the form of `/api/fetch_dogs-jkhj12` where the trailing data is a unique hash. As we update our server functions, the hash will change.

To make server functions maintain a stable endpoint, we can manually name them with the `endpoint = "xyz"` attribute.

src/guide_deploy.rs

```
#[server(endpoint = "list_dogs")]
pub async fn list_dogs() -> Result<Vec<usize, String>, ServerFnError> {
  todo!()
}

#[server(endpoint = "remove_dog")]
pub async fn remove_dog(id: usize) -> Result<(), ServerFnError> {
```

```

    todo!()
}

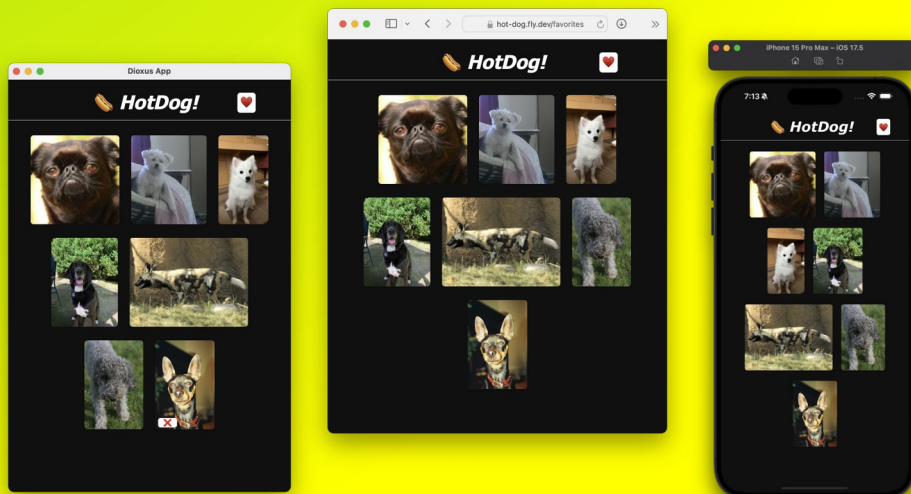
#[server(endpoint = "save_dog")]
pub async fn save_dog(image: String) -> Result<(), ServerFnError> {
    todo!()
}

```

Let's re-deploy our web app with `fly deploy`. This deploy should complete faster thanks to `cargo chef` caching our build.

Now, with `dx serve --platform desktop`, we should be able to interact with the same backend across web and desktop.

Amazing! Our startup is coming along nicely.



Next Steps

Our app isn't done yet, but this guide has become pretty long!

There's so much extra to do:

- Adding users, login, and auth.
- Protecting our site from DDOS with tools Cloudflare.
- Adding more features
- Marketing and sharing with friends!

Wrapping Up

Congrats on making it through our *HotDog* tutorial!

Hopefully this isn't the end of our journey, but rather, a bold new beginning.

Up Next

We strongly recommend moving on to the [Essential Topics](#) to become familiar with the important details of Dioxus.

The essentials section will guide you through key concepts in Dioxus:

- [Building UIs with RSX](#) will teach you how to define html inside your Dioxus app with `rsx`.
- [Component Lifecycle](#) teaches you about the lifecycle of components along with the hooks you need to run code when the component is first created, mounted, and removed.
- [Managing State](#) guides you through how state works in Dioxus. It will teach you how to create state with `use_signal`, derive state with `use_memo`, and integrate state with asynchronous tasks with `use_resource`. Along the way, you will learn about you can use reactivity to declaratively describe your UI.
- [Breaking Out](#) will teach you how to break out of Dioxus' rendering model to run JavaScript or interact with the DOM directly with `web-sys`.

Ideas for New Features

Challenge yourself by adding new features to *HotDog*.

- Add animations to your app with CSS or [dioxus-motion](#).
- Style your app with a library like [TailwindCSS](#).
- Add inputs to allow users to upload their own dog photos.
- Make it social! Add login, user accounts, and a feed.
- Remix HotDog for something entirely new.

FAQ

You might be curious how particular features work in Dioxus. Let's try to address a few popular question:

Is Dioxus Fast?

Dioxus is really fast. Dioxus is built around an *extremely* performant VirtualDom. While a VirtualDom might sound like "[pure overhead](#)", Dioxus leverages compile-time optimizations that make it faster than nearly all UI frameworks.

On the web, Dioxus [is on par with frameworks like Solid and Svelte](#) which leverage new reactivity models or custom compilers.

Is Rust too hard?

Rust is a notoriously difficult language to learn, but it's extremely powerful. Dioxus has been designed to use the "easy" parts of Rust. For example, Dioxus primarily uses single-threaded code and avoids complicated generic interfaces.

All-in-all, the productivity you gain by building once and deploying everywhere usually makes up for the steeper learning curve. Plus, as Dioxus matures, interfaces become easier to use and the ecosystem improves.

Does Dioxus support "xyz"?

- TailwindCSS: Yes, but it requires an additional step, see the [docs for more info](#).

- Native Widgets: Yes, but you might need to write this code yourself. The Rust ecosystem is young.
- Shadcn-ui: Not yet, a radix-ui port to Dioxus is underway but we're not done yet.
- Actix/Rocket: Yes, kind-of. In Dioxus, Server-Functions are built on axum, but you can use server-side-rendering with any framework.
- Static-site-generation: Yes! The docs are currently under construction, so stay tuned. 🚧
- Animations: Yes, you can use CSS animations, [dioxus-motion](#), or integrate with the system animation APIs.
- Accessibility: Yes. Use semantic HTML and aria-tags for good screen-reader support.
- WGPU: Yes, kinda. You can overlay Dioxus on a WGPU scene or experiment with Dioxus-Native which renders with WGPU.
- AI: Yes, you can interface with providers like [OpenAI](#) or even [build your own provider](#).

If you have an FAQ not covered here, suggest an edit!

Why RSX instead of HTML?

If you've seen React's JSX or the `html! {}` Rust macro, you might be curious as to why Dioxus chose to use its own syntax instead of a syntax that looks more similar to HTML.

A few reasons:

- RSX gets token coloring and code-folding without additional tooling
- RSX is faster to type since curly braces are auto-closed
- Not all RSX is HTML - Dioxus can be used in non-HTML contexts
- HTML is not valid Rust - not all HTML can be used in `html! {}`

We want the out-of-the-box experience for Dioxus to be great without setting up any custom tooling.

The Dioxus [VSCode Extension](#) can convert HTML to RSX as well as the CLI with `dx translate`.

Should I use Dioxus or Tauri or Leptos or Iced or...?

There's a budding ecosystem of Rust projects for building apps and websites! We have a good comparison in our [Readme.md](#)

Dioxus is an "all-in-one" solution for building apps with Rust. We try to be helpful every step of the way: getting started, iterating, testing, bundling, and deploying.

- **Tauri:** You can use Dioxus with Tauri, but we focus primarily on the standalone Dioxus experience. Tauri is a good choice if you want to use a JavaScript frontend or need full-featured access to the Webview DOM.
- **Leptos:** Dioxus and Leptos are both good choices for fullstack web development with differences in syntax and reactivity models. Dioxus is architected to support desktop and mobile "1st-class" whereas Leptos is primarily a web framework.
- **Iced:** Iced is GUI framework that renders using its own render engine, making it highly customizable. Iced is a good choice for apps that need access to lower-level rendering through WGPU. Dioxus will be releasing its own WGPU renderer in early 2025, so stay tuned!

Dioxus is quite a large project and has been a bit slower to mature than other projects with larger development teams or smaller scopes. With 0.6, Dioxus is more mature but still has some ground to cover.

Core Topics

This section will guide you through key concepts in Dioxus:

- [Building UIs with RSX](#) will teach you how to define html inside your Dioxus app with `rsx`.
- [Component Lifecycle](#) teaches you about the lifecycle of components along with the hooks you need to run code when the component is first created, mounted, and removed.
- [Managing State](#) guides you through how state works in Dioxus. It will teach you how to create state with `use_signal`, derive state with `use_memo`, and integrate state with asynchronous tasks with `use_resource`. Along the way, you will learn about you can use reactivity to declaratively describe your UI.
- [Breaking Out](#) will teach you how to break out of Dioxus' rendering model to run JavaScript or interact with the DOM directly with `web-sys`.
- [Async](#) will teach you how to integrate async tasks with Dioxus and how to handle loading states while waiting for async tasks to finish.
- [Error Handling](#) will teach you how to throw and handle errors in Dioxus.

Building UIs with RSX

Dioxus renders to HTML, if you are not familiar with HTML, this guide will help you get started with the basics. For more detail, the [MDN docs](#) are a great resource.

Text Nodes

Any content surrounded by quotes is rendered as a text node in `rsx`:

`src/building_uis_with_rsx.rs`

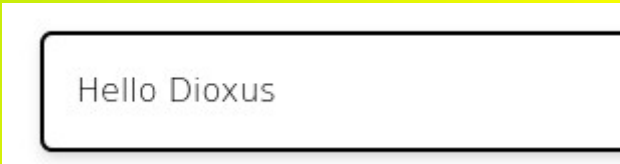
```
rsx! {  
  "Hello world"  
}
```

Hello world

You can include formatted segments inside of the text just like the `format!` macro:

src/building_uis_with_rsx.rs

```
let user = use_signal(|| User {
    name: "Dioxus".to_string(),
});
rsx! {
    // Unlike the format macro, you can include many expressions inline in the
    formatted text
    "Hello {user.read().name}"
}
```



Elements

The most basic building block of HTML is an element. In rsx, you can create elements with the name and then curly braces. One of the most common elements is the `input` element. The input element creates an interactive input box:

src/building_uis_with_rsx.rs

```
rsx! {
    input {}
}
```



Bonus: web components Any element with a dash in the name is a web component. Web components are rendered directly in dioxus without type checking. We recommend wrapping web components in a type safe component to make them easier to use.

src/building_uis_with_rsx.rs

```
rsx! {
    my-web-component {}
}
```

Attributes

Attributes provide extra information about an element. You can specify attributes in dioxus inside an element's braces by typing the name of the attribute, a colon, and then the value (typically a formatted string). We can use an attribute to set the `type` of an input element. The default type is `text` which shows a text input box, but we can set it to `number` to only accept numbers:

src/building_uis_with_rsx.rs

```
rsx! {  
    input { type: "number" }  
}
```

Just like text nodes, attributes can include formatted segments. We can set the value of the input element to a signal to control it:

src/building_uis_with_rsx.rs

```
let mut value = use_signal(|| "Hello world".to_string());  
rsx! {  
    input { value: "{value}" }  
}
```

Conditional Attributes

You can conditionally set an attribute by setting the attribute value to an unterminated if statement. If the if statement evaluates to true, the attribute will be set:

src/building_uis_with_rsx.rs

```
let number_type = use_signal(|| false);  
rsx! {  
    input { type: if number_type() { "number" } }  
}
```

Event Listeners

Event listeners allow you to respond to user input. In `rsx`, event handlers always start with `on`. The syntax is the same as normal attributes, but event handlers only accept a closure that responds to the event. We can attach an event listener to the `oninput` event of the `input` element to listen for changes to the input:

`src/building_uis_with_rsx.rs`

```
let mut value = use_signal(|| "Hello world".to_string());
rsx! {
  input {
    oninput: move |event| value.set(event.value()),
    value: "{value}"
  }
}
```

Children

You can add children to an element after all attributes and event listeners. Elements can accept text, components or other elements as children. We can add a `div` element around our input to center it:

`src/building_uis_with_rsx.rs`

```
rsx! {
  div {
    // display sets the layout mode of the element
    display: "flex",
    // justify-content centers the element horizontally
    justify_content: "center",
    input {
      type: "number"
    }
  }
}
```

Loops

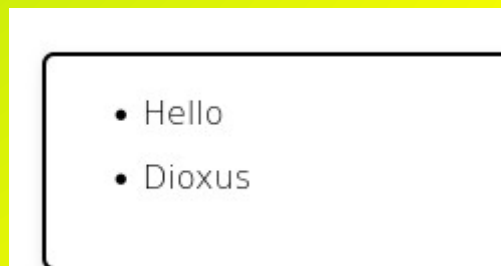
You can insert for loops directly in `rsx`. The body of the loop accepts any number of children that will be rendered with each iteration of the loop. The `ul` element in `html` renders an unordered list

with any number of `li` (list item) elements. We can use those two elements to render a list of items in a loop:

src/building_uis_with_rsx.rs

```
let mut items = use_signal(|| vec!["Hello", "Dioxus"]);

rsx! {
  ul {
    for item in items.iter() {
      li { "{item}" }
    }
  }
}
```



Each item in your list should have unique value that is stable across rerenders called a key. Keys are used to identify how items move while diffing. Without keys, it is easy to accidentally lose or move state when you reorder items in a list. We can add keys to our list items by using the `key` attribute:

src/building_uis_with_rsx.rs

```
let mut items = use_signal(|| vec!["Hello", "Dioxus"]);

rsx! {
  ul {
    for item in items.iter() {
      li { key: "{item}", "{item}" }
    }
  }
}
```

If Statements

You can also use if/else statements in rsx. Each branch of the if statement accepts child nodes that will be rendered if the condition is true. We can use the `if` statement to conditionally render a login screen:

src/building_uis_with_rsx.rs

```
let logged_in = use_signal(|| false);

rsx! {
  div {
    if logged_in() {
      "You are logged in"
    } else {
      "You are not logged in"
    }
  }
}
```

```
}
```

You are not logged in

Why RSX and not HTML ?

If you've seen React's JSX or the `html! {}` Rust macro, you might be curious as to why Dioxus chose to use its own syntax instead of a syntax that looks more similar to HTML.

A few reasons:

- RSX gets token coloring and code-folding without additional tooling
- RSX is faster to type since curly braces are auto-closed
- Not all RSX is HTML - Dioxus can be used in non-HTML contexts
- HTML is not valid Rust - not all HTML can be used in `html! {}`

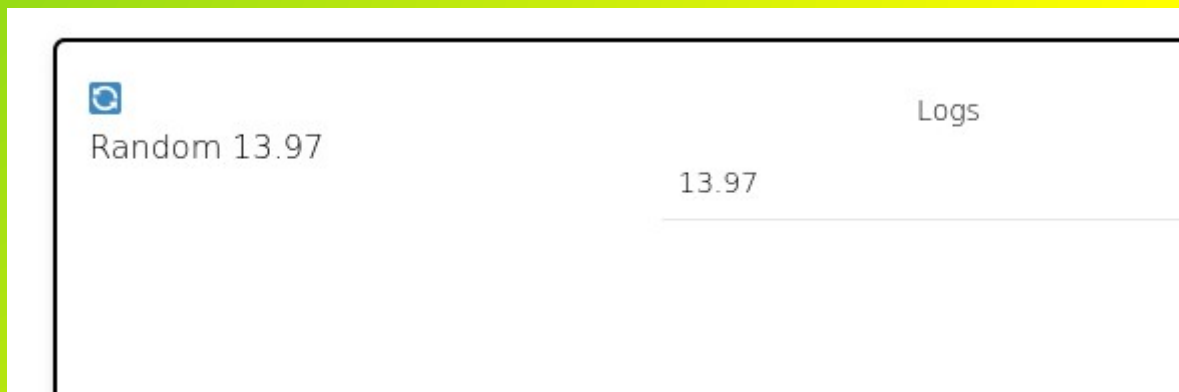
Component Lifecycle

Initializing State with use_hook

`use_hook` lets you create new state for your component. The closure you pass to `use_hook` will be called once the first time the component is rendered. Every time the component is re-rendered, the value that was created the first run will be re-used.

src/component_lifecycle.rs

```
fn UseHook() -> Element {  
    // The closure that is passed to use_hook will be called once the first time  
the component is rendered  
    let random_number = use_hook(|| {  
        let new_random_number = random_number();  
  
        log!("{new_random_number}");  
  
        new_random_number  
    });  
  
    rsx! {  
        div { "Random {random_number}" }  
    }  
}
```



Rerendering

You can use [tracked values](#) to re-render your component whenever a value changes.

src/component_lifecycle.rs

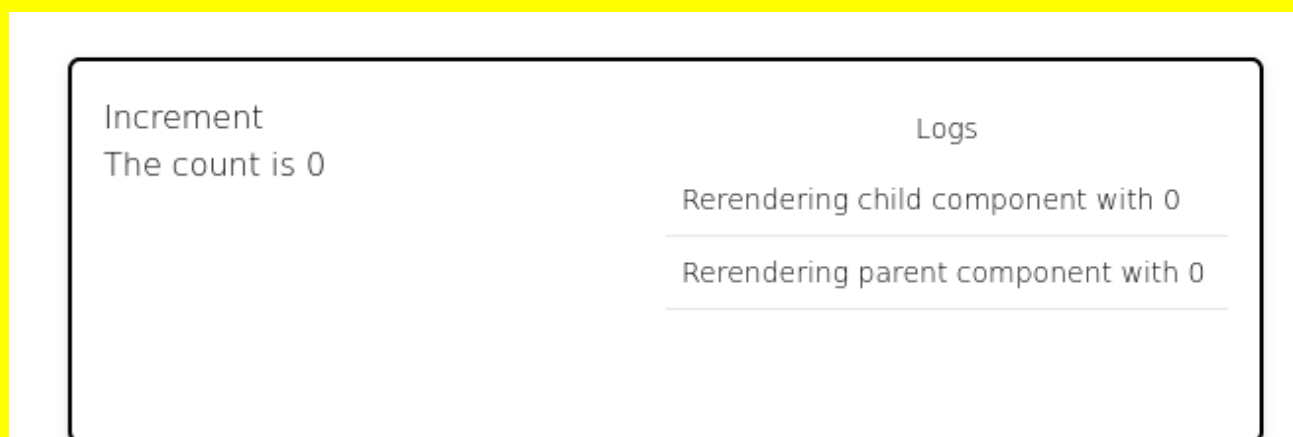
```
fn Rerenders() -> Element {
    let mut count = use_signal(|| 0);

    log!("Rerendering parent component with {}", *count.peek());

    rsx! {
        button { onclick: move |_| count += 1, "Increment" }
        // Since we read count here, the component will rerender when count
changes
        Count { current_count: count() }
    }
}

// If the count prop changes, the component will rerender
#[component]
fn Count(current_count: i32) -> Element {
    log!("Rerendering child component with {current_count}");

    rsx! {
        div { "The count is {current_count}" }
    }
}
```



⚠ Don't mutate state in the body of a component

You should avoid changing state in the body of a component. If you read and write to state in the body of a component, you can cause an infinite loop as the component tries to rerender because of the change which triggers another state change.

src/component_lifecycle.rs

```
fn Bad() -> Element {
  let mut count = use_signal(|| 0);

  // ✗ Don't mutate state in the body of the component.
  // It can easily cause an infinite loop!
  count += 1;

  rsx! { "{count}" }
}
```

Instead, derive state with `use_memo`, `use_resource`, or mutate state in a effect.

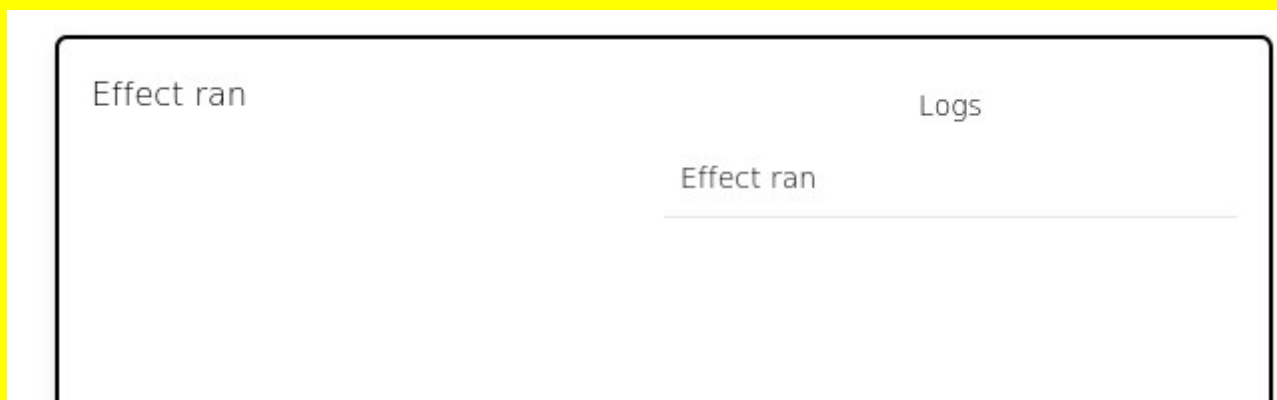
Using Effects

You can use effects to run code whenever a component is rendered.

src/component_lifecycle.rs

```
fn Effect() -> Element {
  // Effects run after the component is rendered
  // You can use them to read or modify the rendered component
  use_effect(|| {
    log!("Effect ran");
    document::eval(&format!(
      "document.getElementById('effect-output').innerText = 'Effect ran'"
    ));
  });

  rsx! {
    div { id: "effect-output", "This will be changed by the effect" }
  }
}
```



Cleaning Up Components with Drop

Before a component is dropped, it will drop all of its hooks. You can use this drop behavior to clean up any resources that your component is using. If you just need the drop effect, you can use the [use_drop](#) hook.

src/component_lifecycle.rs

```
fn TogglesChild() -> Element {
    let mut show = use_signal(|| true);

    rsx! {
        button { onclick: move |_| show.toggle(), "Toggle" }
        if show() {
            Child {}
        }
    }
}

fn Child() -> Element {
    // You can use the use_drop hook to clean up any resources
    use_drop(|| {
        log!("Child dropped");
    });

    rsx! {
        div { "Child" }
    }
}
```



Managing State

In Dioxus, your app is defined as a function of the current state. As the state changes, the parts of your app that depend on that state will automatically re-run. Reactivity automatically tracks state and updates derived state in your application.

Creating State

You can create mutable state in Dioxus with Signals. Signals are tracked values that automatically update your app when you change them. They form the skeleton of your app's state from which you can derive other state. Signals are often driven directly from user input through event handlers or async tasks.

You can create a signal with the `use_signal` hook:

src/reactivity.rs

```
let mut signal = use_signal(|| 0);
```

Once you have your signal, you can clone it by calling the signal like a function or get a reference to the inner value with the `.read()` method:

src/reactivity.rs

```
// Call the signal like a function to clone the current value
let value: i32 = signal();
// get a reference to the inner value with the .read() method
let value: &i32 = &signal.read();
// or use one of the traits implemented for Signal like Display
log!("{signal}");
```

Finally, you can set the value of the signal with the `.set()` method or get a mutable reference to the inner value with the `.write()` method:

src/reactivity.rs

```
// Set the value from the signal
signal.set(1);
// get a mutable reference to the inner value with the .write() method
let mut value: &mut i32 = &mut signal.write();
*value += 1;
```

Reactive Scopes

The simplest reactive primitive in Dioxus is the `use_effect` hook. It creates a closure that is run any time a tracked value that is run inside the closure changes.

Any value you read inside the closure will become a dependency of the effect. If the value changes, the effect will rerun.

src/reactivity.rs

```
fn Effect() -> Element {
    // use_signal creates a tracked value called count
    let mut count = use_signal(|| 0);

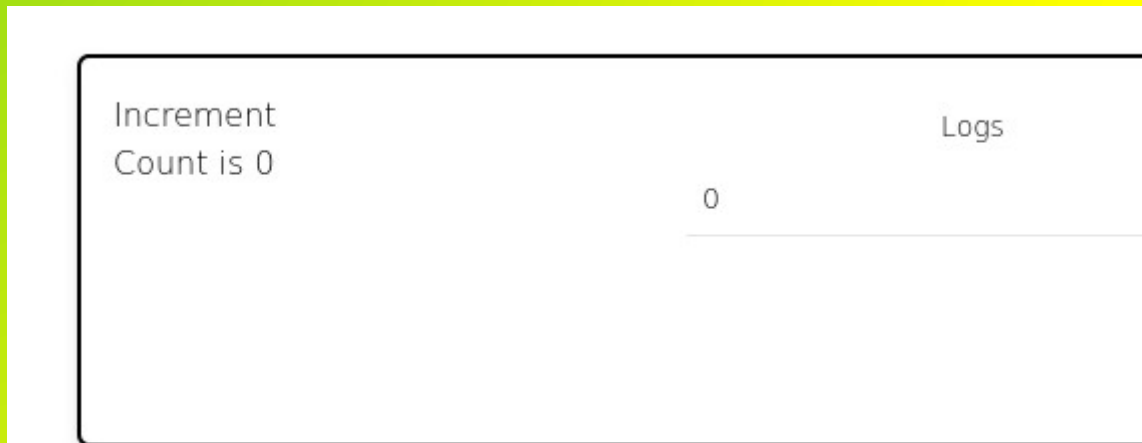
    use_effect(move || {
        // When we read count, it becomes a dependency of the effect
        let current_count = count();
        // Whenever count changes, the effect will rerun
        log!("{current_count}");
    });

    rsx! {
        button { onclick: move |_| count += 1, "Increment" }
    }
}
```

```

    div { "Count is {count}" }
  }
}

```



Derived State

`use_memo` is a reactive primitive that lets you derive state from any tracked value. It takes a closure that computes the new state and returns a tracked value with the current state of the memo. Any time a dependency of the memo changes, the memo will rerun.

The value you return from the closure will only change when the output of the closure changes (`PartialEq` between the old and new value returns false).

src/reactivity.rs

```

fn Memo() -> Element {
  let mut count = use_signal(|| 0);

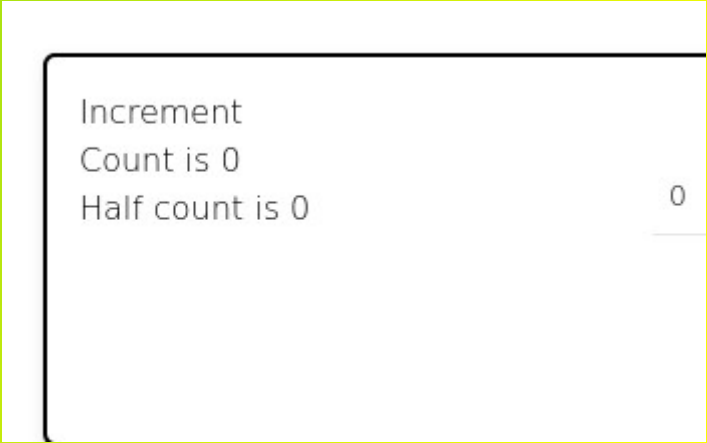
  // use_memo creates a tracked value that is derived from count
  // Since we read count inside the closure, it becomes a dependency of the
memo
  // Whenever count changes, the memo will rerun
  let half_count = use_memo(move || count() / 2);

  use_effect(move || {
    // half_count is itself a tracked value
    // When we read half_count, it becomes a dependency of the effect
    // and the effect will rerun when half_count changes
    log!("{half_count}");
  });

  rsx! {
    button { onclick: move |_| count += 1, "Increment" }

    div { "Count is {count}" }
    div { "Half count is {half_count}" }
  }
}

```



Increment
Count is 0
Half count is 0

0

Derived Async State

`use_resource` is a reactive primitive that lets you derive state from any async closure. It takes an async closure that computes the new state and returns a tracked value with the current state of the resource. Any time a dependency of the resource changes, the resource will rerun.

The value you return from the closure will only change when the state of the future changes. Unlike `use_memo`, the resource's output is not memoized with `PartialEq`.

src/reactivity.rs

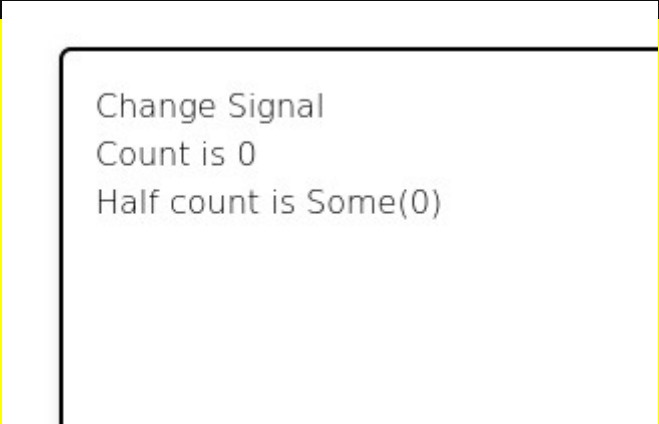
```
fn Resource() -> Element {
    let mut count = use_signal(|| 0);

    // use_resource creates a tracked value that is derived from count
    // Since we read count inside the closure, it becomes a dependency of the
    resource
    // Whenever count changes, the resource will rerun
    let half_count = use_resource(move || async move {
        // You can do async work inside resources
        gloo_timers::future::TimeoutFuture::new(100).await;
        count() / 2
    });

    use_effect(move || {
        // half_count is itself a tracked value
        // When we read half_count, it becomes a dependency of the effect
        // and the effect will rerun when half_count changes
        log!("{:?}", half_count());
    });

    rsx! {
        button { onclick: move |_| count += 1, "Change Signal" }

        div { "Count is {count}" }
        div { "Half count is {half_count():?}" }
    }
}
```



Change Signal
Count is 0
Half count is Some(0)

Derived UI

Components are functions that return some UI. They memorize the output of the function just like memos. Components keep track of any dependencies you read inside the component and rerun when those dependencies change.

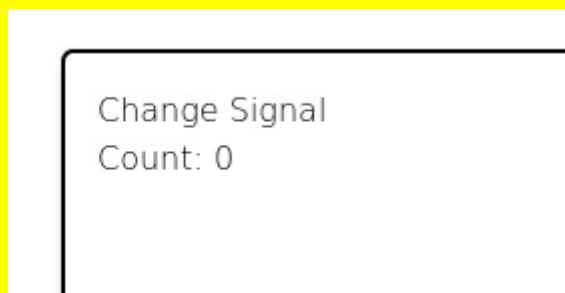
src/reactivity.rs

```
fn Component() -> Element {
    let mut count = use_signal(|| 0);

    rsx! {
        button { onclick: move |_| count += 1, "Change Signal" }

        // Since we read count inside Component, it becomes a dependency of
        // Whenever count changes, Component will rerun
        Count { count: count() }
    }
}

// Components automatically memorize their props. If the props change, Count
// will rerun
#[component]
fn Count(count: i32) -> Element {
    rsx! {
        div { "Count: {count}" }
    }
}
```



Working with Untracked State

Most of the state in your app will be tracked values. All built in hooks return tracked values, and we encourage custom hooks to do the same. However, there are times when you need to work with untracked state. For example, you may receive a raw untracked value in props. When you read an untracked value inside a reactive context, it will not subscribe to the value:

src/reactivity.rs

```
fn Component() -> Element {
```

```

let mut count = use_signal(|| 0);

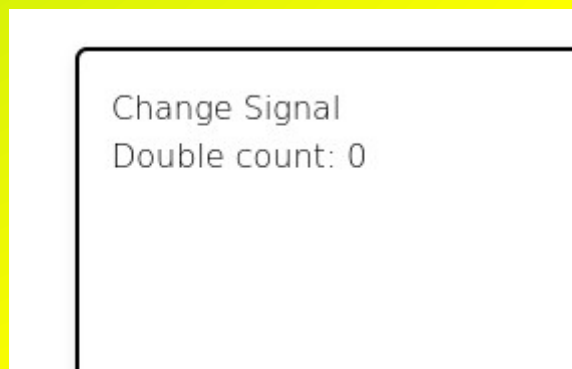
rsx! {
  button { onclick: move |_| count += 1, "Change Signal" }

  Count { count: count() }
}

// The count reruns the component when it changes, but it is not a tracked value
#[component]
fn Count(count: i32) -> Element {
  // When you read count inside the memo, it does not subscribe to the count
  signal
  // because the value is not reactive
  let double_count = use_memo(move || count * 2);

  rsx! {
    div { "Double count: {double_count}" }
  }
}

```



You can start tracking raw state with the `use_reactive` hook. This hook takes a tuple of dependencies and returns a reactive closure. When the closure is called in a reactive context, it will track subscribe to the dependencies and rerun the closure when the dependencies change.

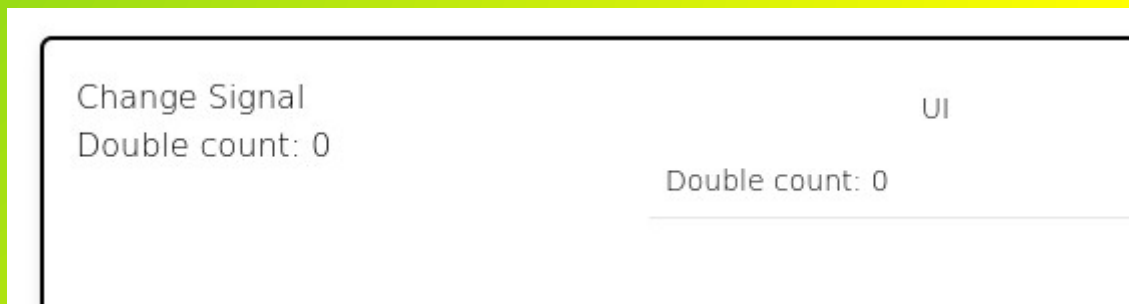
src/reactivity.rs

```

#[component]
fn Count(count: i32) -> Element {
  // You can manually track a non-reactive value with the use_reactive hook
  let double_count = use_memo(
    // Use reactive takes a tuple of dependencies and returns a reactive
    closure
    use_reactive!(|(count,)| count * 2),
  );

  rsx! {
    div { "Double count: {double_count}" }
  }
}

```



Making Props Reactive

To avoid losing reactivity with props, we recommend you wrap any props you want to track in a `ReadOnlySignal`. Dioxus will automatically convert `T` into `ReadOnlySignal<T>` when you pass props to the component. This will ensure your props are tracked and rerun any state you derive in the component:

src/reactivity.rs

```
// You can track props by wrapping the type in a ReadOnlySignal
// Dioxus will automatically convert T into ReadOnlySignal<T> when you pass
// props to the component
#[component]
fn Count(count: ReadOnlySignal<i32>) -> Element {
    // Then when you read count inside the memo, it subscribes to the count
    signal
    let double_count = use_memo(move || count() * 2);

    rsx! {
        div { "Double count: {double_count}" }
    }
}
```

Moving Around State

As you create signals and derived state in your app, you will need to move around that state between components. Dioxus provides three different ways to pass around state:

Passing props

You can pass your values through component [props](#). This should be your default when passing around state. It is the most explicit and local to your component. Use this until it gets annoying to pass around the value:

src/moving_state_around.rs

```
pub fn ParentComponent() -> Element {
    let count = use_signal(|| 0);

    rsx! {
        "Count is {count}"
        IncrementButton {
            count
        }
    }
}
```

```

}

#[component]
fn IncrementButton(mut count: Signal<i32>) -> Element {
    rsx! {
        button {
            onclick: move |_| count += 1,
            "Increment"
        }
    }
}

```



Passing context

If you need a slightly more powerful way to pass around state, you can use the context API.

The context API lets you pass state from a parent component to all children. This is useful if you want to share state between many components. You can insert a unique type into the context with the [use_context_provider](#) hook in the parent component. Then you can access the context in any child component with the [use_context](#) hook.

src/moving_state_around.rs

```

#[derive(Clone, Copy)]
struct MyState {
    count: Signal<i32>,
}

pub fn ParentComponent() -> Element {
    // Use context provider provides an unique type to all children of this
    // component
    let state = use_context_provider(|| MyState {
        count: Signal::new(0),
    });

    rsx! {
        "Count is {state.count}"
        // IncrementButton will have access to the count without explicitly
        // passing it through props
        IncrementButton {}
    }
}

#[component]
fn IncrementButton() -> Element {
    // Use context gets the value from a parent component
    let mut count = use_context::<MyState>().count;

    rsx! {
        button {
            onclick: move |_| count += 1,
            "Increment"
        }
    }
}

```

```
}  
}
```



This is slightly less explicit than passing it as a prop, but it is still local to the component. This is really great if you want state that is global to part of your app. It lets you create multiple global-ish states while still making state different when you reuse components. If I create a new `ParentComponent`, it will have a new `MyState`.

Using globals

Finally, if you have truly global state, you can put your state in a `Global<T>` static. This is useful if you want to share state with your whole app:

src/moving_state_around.rs

```
use dioxus::prelude::*;  
// Globals are created the first time you access them with the closure you pass  
// to Global::new  
static COUNT: GlobalSignal<i32> = Global::new(|| 0);  
  
pub fn ParentComponent() -> Element {  
    rsx! {  
        "Count is {COUNT}"  
        IncrementButton {}  
    }  
}  
  
fn IncrementButton() -> Element {  
    rsx! {  
        button {  
            // You don't need to pass anything around or get anything out of the  
            // context because COUNT is global  
            onclick: move |_| *COUNT.write() += 1,  
            "Increment"  
        }  
    }  
}
```

Global state can be very ergonomic if your state is truly global, but you shouldn't use it if you need state to be different for different instances of your component. If I create another `IncrementButton` it will use the same `COUNT`. Libraries should generally avoid this to make components more reusable.

Note: Even though it is in a static, `COUNT` will be different for each app instance so you don't need to worry about state mangling when multiple instances of your app are running on the server

Handling Asynchronous Tasks

Asynchronous tasks are a core part of any modern application. Dioxus provides a few different methods to handle asynchronous tasks. This guide will cover how to use each of them. If you already know what kind of asynchronous task you need, you can skip to the section for that task:

- [spawn](#) is great for futures you need to run in the background that don't return a value
- [use_resource](#) handles asynchronous state while retaining control of exactly what happens while the future is running
- It can be combined with [Suspense](#) to handle many pending tasks with the same loading view

Running Futures with spawn

The [spawn](#) method spawns a future in the background and returns a `Task` that you can use to cancel the future. Spawn is great for futures you want to start and then forget about like sending analytics data to a server:

src/asynchronous.rs

```
let mut response = use_signal(|| "Click to start a request".to_string());

rsx! {
  button {
    onclick: move |_| {
      response.set("...".into());
      // Spawn will start a task running in the background
      spawn(async move {
        let resp = request::Client::new()
          .get("https://dioxuslabs.com")
          .send()
          .await;

        if resp.is_ok() {
          response.set("dioxuslabs.com responded!".into());
        } else {
          response.set("failed to fetch response!".into());
        }
      });
    },
    "{response}"
  }
}
```

Since spawning in event handlers is very common, Dioxus provides a more concise syntax for async event handlers. If you return a future from an event handler, Dioxus will automatically spawn it:

src/asynchronous.rs

```
let mut response = use_signal(|| "Click to start a request".to_string());

rsx! {
  button {
    // Async closures passed to event handlers are automatically spawned
    onclick: move |_| async move {
      response.set("...".into());
    }
  }
}
```

```

        let resp = request::Client::new()
        .get("https://dioxuslabs.com")
        .send()
        .await;

        if resp.is_ok() {
            response.set("dioxuslabs.com responded!".into());
        } else {
            response.set("failed to fetch response!".into());
        }
    },
    "{response}"
}
}

```

The future you pass to the `spawn` will automatically be cancelled when the component is unmounted. If you need to keep the future running until it is finished, you can use [`spawn\_forever`](#) instead.

Asynchronous State with `use_resource`

The `use_resource` can be used to derive asynchronous state. It takes an async closure to calculate the state and returns a tracked value with the current state of the future. Any time a dependency of the resource changes, the resource will rerun:

src/asynchronous.rs

```

let mut breed = use_signal(|| "hound".to_string());
let dogs = use_resource(move || async move {
    request::Client::new()
        // Since breed is read inside the async closure, the resource will
        subscribe to the signal
        // and rerun when the breed is written to
        .get(format!("https://dog.ceo/api/breed/{breed}/images"))
        .send()
        .await?
        .json::<BreedResponse>()
        .await
});

rsx! {
    input {
        value: "{breed}",
        // When the input is changed and the breed is set, the resource will
        rerun
        oninput: move |evt| breed.set(evt.value()),
    }

    div {
        display: "flex",
        flex_direction: "row",
        // You can read resource just like a signal. If the resource is still
        // running, it will return None
        if let Some(response) = &*dogs.read() {
            match response {
                Ok(urls) => rsx! {
                    for image in urls.iter().take(3) {
                        img {
                            src: "{image}",
                            width: "100px",

```

```

        height: "100px",
      }
    },
    Err(err) => rsx! { "Failed to fetch response: {err}" },
  } else {
    "Loading..."
  }
}
}

```

The `use_resource` hook might look similar to the `use_memo` hook, but unlike `use_memo`, the resource's output is not memoized with `PartialEq`. That means any components/reactive hooks that read the output will rerun if the future reruns even if the value it returns is the same:

src/asynchronous.rs

```

let mut number = use_signal(|| 0);

// Resources rerun any time their dependencies change. They will
// rerun any reactive scopes that read the resource when they finish
// even if the value hasn't changed
let halved_resource = use_resource(move || async move { number() / 2 });

log!("Component reran");

rsx! {
  button {
    onclick: move |_| number += 1,
    "Increment"
  }
  p {
    if let Some(halved) = halved_resource() {
      "Halved: {halved}"
    } else {
      "Loading..."
    }
  }
}

```

Note: The future you pass to `use_resource` must be cancel safe. Cancel safe futures are futures that can be stopped at any await point without causing issues. For example, this task is not cancel safe:

src/asynchronous.rs

```

static RESOURCES_RUNNING: GlobalSignal<HashSet<String>> = Signal::global(||
HashSet::new());
let mut breed = use_signal(|| "hound".to_string());
let dogs = use_resource(move || async move {
  // Modify some global state
  RESOURCES_RUNNING.write().insert(breed());

  // Wait for a future to finish. The resource may cancel
  // without warning if breed is changed while the future is running. If
  // it does, then the breed pushed to RESOURCES_RUNNING will never be popped
  let response = request::Client::new()
    .get(format!("https://dog.ceo/api/breed/{breed}/images"))
    .send()
    .await?

```

```

        .json::<BreedResponse>()
        .await;

    // Restore some global state
    RESOURCES_RUNNING.write().remove(&breed());

    response
});

```

It can be fixed by making sure the global state is restored when the future is dropped:

src/asynchronous.rs

```

static RESOURCES_RUNNING: GlobalSignal<HashSet<String>> = Signal::global(||
HashSet::new());
let mut breed = use_signal(|| "hound".to_string());
let dogs = use_resource(move || async move {
    // Modify some global state
    RESOURCES_RUNNING.write().insert(breed());

    // Automatically restore the global state when the future is dropped, even
if
    // isn't finished
    struct DropGuard(String);
    impl Drop for DropGuard {
        fn drop(&mut self) {
            RESOURCES_RUNNING.write().remove(&self.0);
        }
    }
    let _guard = DropGuard(breed());

    // Wait for a future to finish. The resource may cancel
    // without warning if breed is changed while the future is running. If
    // it does, then it will be dropped and the breed will be popped
    request::Client::new()
        .get(format!("https://dog.ceo/api/breed/{breed}/images"))
        .send()
        .await?
        .json::<BreedResponse>()
        .await
});

```

Async methods will often mention if they are cancel safe in their documentation.

Unified Loading Views with suspense

`SuspenseBoundary` is a convenient way to bundle multiple async tasks into a single loading view. It accepts a loading closure and children. You can suspend tasks in children to pause rendering of that child until the future is finished. The suspense boundary will show the loading view instead of the children while any of its children are suspended. Once that suspense is resolved, it will show the children again.

We can use a suspense boundary to show a grid of different breeds of dogs without handling each loading state individually:

src/asynchronous.rs

```

fn DogGrid() -> Element {
    rsx! {

```

```

SuspenseBoundary {
    // When any child components (like BreedGallery) are suspended, this
closure will
    // be called and the loading view will be rendered instead of the
children
    fallback: |_| rsx! {
        div {
            width: "100%",
            height: "100%",
            display: "flex",
            align_items: "center",
            justify_content: "center",
            "Loading..."
        }
    },
    div {
        display: "flex",
        flex_direction: "column",
        BreedGallery {
            breed: "hound"
        }
        BreedGallery {
            breed: "poodle"
        }
        BreedGallery {
            breed: "beagle"
        }
    }
}
}
}

#[component]
fn BreedGallery(breed: ReadOnlySignal<String>) -> Element {
    let response = use_resource(move || async move {
        // Artificially slow down the request to make the loading indicator
easier to seer
        gloo_timers::future::TimeoutFuture::new(1000).await;
        request::Client::new()
            .get(format!("https://dog.ceo/api/breed/{breed}/images"))
            .send()
            .await?
            .json::<BreedResponse>()
            .await
    })
    // Calling .suspend()? will suspend the component and return early while the
future is running
    .suspend()?;

    // Then you can just handle the happy path with the resolved future
    rsx! {
        div {
            display: "flex",
            flex_direction: "row",
            match &*response.read() {
                Ok(urls) => rsx! {
                    for image in urls.iter().take(3) {
                        img {
                            src: "{image}",
                            width: "100px",
                            height: "100px",
                        }
                    }
                }
            }
        }
    },

```

```

        Err(err) => rsx! { "Failed to fetch response: {err}" },
    }
}
}
}

```

If you need to change the loading view while a specific task is loading, you can provide a different loading view with the `with_loading_placeholder` method. The loading placeholder you return from the method will be passed to the suspense boundary and may choose to render it instead of the default loading view:

src/asynchronous.rs

```

fn DogGrid() -> Element {
    rsx! {
        SuspenseBoundary {
            // The fallback closure accepts a SuspenseContext which contains
            // information about the suspended component
            fallback: |suspense_context: SuspenseContext| if let Some(view) =
suspense_context.suspense_placeholder() {
                view
            } else {
                rsx! {
                    div {
                        width: "100%",
                        height: "100%",
                        display: "flex",
                        align_items: "center",
                        justify_content: "center",
                        "Loading..."
                    }
                }
            },
            div {
                display: "flex",
                flex_direction: "column",
                BreedGallery {
                    breed: "hound"
                }
                BreedGallery {
                    breed: "poodle"
                }
                BreedGallery {
                    breed: "beagle"
                }
            }
        }
    }
}

#[component]
fn BreedGallery(breed: ReadOnlySignal<String>) -> Element {
    let response = use_resource(move || async move {
        gloo_timers::future::TimeoutFuture::new(breed().len() as u32 *
100).await;
        request::Client::new()
            .get(format!("https://dog.ceo/api/breed/{breed}/images"))
            .send()
            .await?
            .json::<BreedResponse>()
            .await
    })
}

```

```

.suspend()
// You can pass up a loading placeholder to the nearest SuspenseBoundary
// with the with_loading_placeholder method
.with_loading_placeholder(move || {
    rsx! {
        div {
            width: "100%",
            height: "100%",
            display: "flex",
            align_items: "center",
            justify_content: "center",
            "Loading {breed}..."
        }
    }
})?;

// Then you can just handle the happy path with the resolved future
rsx! {
    div {
        display: "flex",
        flex_direction: "row",
        match &*response.read() {
            Ok(urls) => rsx! {
                for image in urls.iter().take(3) {
                    img {
                        src: "{image}",
                        width: "100px",
                        height: "100px",
                    }
                }
            },
            Err(err) => rsx! { "Failed to fetch response: {err}" },
        }
    }
}
}

```

Suspense with Fullstack

To use suspense in your fullstack application, you need to use the `use_server_future` hook instead of `use_resource`. `use_server_future` handles serialization of the result of the future for hydration. It will also suspend automatically, so you don't need to call `.suspend()` on the future.

src/asynchronous.rs

```

#[component]
fn BreedGallery(breed: ReadOnlySignal<String>) -> Element {
    // use_server_future is very similar to use_resource, but the value returned
    // from the future
    // must implement Serialize and Deserialize and it is automatically
    // suspended
    let response = use_server_future(move || async move {
        // The future will run on the server during SSR and then get sent to the
        // client
        request::Client::new()
            .get(format!("https://dog.ceo/api/breed/{breed}/images"))
            .send()
            .await
        // request::Result does not implement Serialize, so we need to map
        // it to a string which
    })
}

```

```

        // can be serialized
        .map_err(|err| err.to_string())?
        .json::()
        .await
        .map_err(|err| err.to_string())
    // use_server_future calls `suspend` internally, so you don't need to
    call it manually, but you
    // do need to bubble up the suspense variant with `?`
    })?;

    // If the future was still pending, it would have returned suspended with
    the `?` above
    // we can unwrap the None case here to get the inner result
    let response_read = response.read();
    let response = response_read.as_ref().unwrap();

    // Then you can just handle the happy path with the resolved future
    rsx! {
        div {
            display: "flex",
            flex_direction: "row",
            match response {
                Ok(urls) => rsx! {
                    for image in urls.iter().take(3) {
                        img {
                            src: "{image}",
                            width: "100px",
                            height: "100px",
                        }
                    }
                },
                Err(err) => rsx! { "Failed to fetch response: {err}" },
            }
        }
    }
}

```

Unlike `use_resource`, `use_server_future` is only reactive in the closure, not the future itself. If you need to subscribe to another reactive value, you need to read it in the closure before passing it to the future:

`src/asynchronous.rs`

```

let id = use_signal(|| 0);
// ✗ The future inside of use_server_future is not reactive
use_server_future(move || {
    async move {
        // But the future is not reactive which means that the future will not
        subscribe to any reads here
        println!("{id}");
    }
});
// ✓ The closure that creates the future for use_server_future is reactive
use_server_future(move || {
    // The closure itself is reactive which means the future will subscribe to
    any signals you read here
    let cloned_id = id();
    async move {
        // But the future is not reactive which means that the future will not
        subscribe to any reads here
        println!("{cloned_id}");
    }
});

```

When you use `suspense` with `fullstack` without streaming enabled, dioxus will wait until all suspended futures are resolved before sending the resolved html to the client. If you [enable](#) out of order streaming, dioxus will send the finished HTML chunks to the client one at a time as they are resolved:

src/asynchronous.rs

```
fn main() {
    dioxus::LaunchBuilder::new()
        .with_context(server_only! {
            // Enable out of order streaming during SSR
        })
    dioxus::fullstack::ServeConfig::builder().enable_out_of_order_streaming()
        .launch(DogGrid);
}
```

Conclusion

This guide has covered the basics of asynchronous tasks in Dioxus. More detailed documentation about specific hooks are available in docs.rs:

- [use_resource](#)
- [use_server_future](#)
- [SuspenseBoundary](#)
- [spawn](#)
- [spawn_forever](#)

More examples of futures and asynchronous tasks are available in the [example folder](#) in the dioxus repo.

Breaking Out of Dioxus

Dioxus makes it easy to build reactive user interfaces. However, there are some cases where you may need to break out of the reactive paradigm to interact with the DOM directly.

Interacting with JavaScript with eval and web-sys

Dioxus exposes a limited number of [web apis](#) with a nicer interface. If you need access to more APIs, you can use the `eval` function to run JavaScript in the browser.

For example, you can use the `eval` function to read the domain of the current page:

src/breaking_out.rs

```
pub fn Eval() -> Element {
    let mut domain = use_signal(String::new);
    rsx! {
        button {
            // When you click the button, some javascript will run in the
            browser
            // to read the domain and set the signal
            onclick: move |_| async move {
```

```

        domain.set(document::eval("return
document.domain").await.unwrap().to_string());
    },
    "Read Domain"
}
"Current domain: {domain}"
}
}

```

If you are only targeting web, you can also use the [web-sys](#) crate for typed access to the web APIs. Here is what reading the domain looks like with web-sys:

src/breaking_out.rs

```

use ::web_sys::window;
use wasm_bindgen::JsCast;
pub fn WebSys() -> Element {
    let mut domain = use_signal(String::new);
    rsx! {
        button {
            // When you click the button, we use web-sys to read the domain and
a signal
            onclick: move |_| {
                domain
                    .set(
                        window()
                            .unwrap()
                            .document()
                            .unwrap()
                            .dyn_into::<::web_sys::HtmlDocument>()
                            .unwrap()
                            .domain(),
                    );
            },
            "Read Domain"
        }
        "Current domain: {domain}"
    }
}

```

Synchronizing DOM updates with use_effect

If you do need to interact with the DOM directly, you should do so in a `use_effect` hook. This hook will run after the component is rendered and all of the Dioxus UI has been rendered. You can read or modify the DOM in this hook.

For example, you can use the `use_effect` hook to write to a canvas element after it is created:

src/breaking_out.rs

```

pub fn Canvas() -> Element {
    let mut count = use_signal(|| 0);

    use_effect(move || {
        // Effects are reactive like memos, and resources. If you read a value
inside the effect, the effect will rerun when that value changes
        let count = count.read();

        // You can use the count value to update the DOM manually
        document::eval(&format!(

```

```

        r#"var c = document.getElementById("dioxus-canvas");
var ctx = c.getContext("2d");
ctx.clearRect(0, 0, c.width, c.height);
ctx.font = "30px Arial";
ctx.fillText("{count}", 10, 50);"#
    ));
});

    rsx! {
        button {
            // When you click the button, count will be incremented and the
effect will rerun
            onclick: move |_| count += 1,
            "Increment"
        }
        canvas { id: "dioxus-canvas" }
    }
}

```

Getting access to elements with onmounted

If you need a handle to an element that is rendered by dioxus, you can use the `onmounted` event. This event will fire after the element is first mounted to the DOM. It returns a live reference to the element with some methods to interact with it.

You can use the `onmounted` event to do things like focus or scroll to an element after it is rendered:

src/breaking_out.rs

```

pub fn OnMounted() -> Element {
    let mut input_element = use_signal(|| None);

    rsx! {
        div { height: "100px",
            button {
                class: "focus:outline-2 focus:outline-blue-600 focus:outline-
dashed",
                // The onmounted event will run the first time the button
element is mounted
                onmounted: move |element|
input_element.set(Some(element.data())),
                "First button"
            }

            button {
                // When you click the button, if the button element has been
mounted, we focus to that element
                onclick: move |_| async move {
                    if let Some(header) = input_element() {
                        let _ = header.set_focus(true).await;
                    }
                },
                "Focus first button"
            }
        }
    }
}

```

Down casting web sys events

Dioxus provides platform agnostic wrappers over each event type. These wrappers are often nicer to interact with than the raw event types, but they can be more limited. If you are targeting web, you can downcast the event with the `as_web_event` method to get the underlying web-sys event:

src/breaking_out.rs

```
pub fn Downcast() -> Element {
    let mut event_text = use_signal(|| 0);

    rsx! {
        div {
            onmousemove: move |event| {
                #[cfg(feature = "web")]
                {
                    use dioxus::web::WebEventExt;
                    event_text.set(event.as_web_event().movement_x());
                }
            },
            "movement_x was {event_text}"
        }
    }
}
```

movement_x was 0

Error handling

A selling point of Rust for web development is the reliability of always knowing where errors can occur and being forced to handle them. Dioxus provides `ErrorBoundaries` to help you handle errors in a declarative way. This guide will teach you how to use `ErrorBoundaries` and other error handling strategies in Dioxus.

Returning Errors from Components

Astute observers might have noticed that `Element` is actually a type alias for `Result<VNode, RenderError>`. The `RenderError` type can be created from an error type that implements `Error`. You can use `?` to bubble up any errors you encounter while rendering to the nearest error boundary:

src/error_handling.rs

```
#[component]
fn ThrowsError() -> Element {
    // You can return any type that implements `Error`
    let number: i32 = use_hook(|| "1.234").parse()?;

    todo!()
}
```

Capturing errors with ErrorBoundaries

When you return an error from a component, it gets sent to the nearest error boundary. That error boundary can then handle the error and render a fallback UI with the `handle_error` closure:

src/error_handling.rs

```
#[component]
fn Parent() -> Element {
    rsx! {
        ErrorBoundary {
            // The error boundary accepts a closure that will be rendered when
            // an error is thrown in any
            // of the children
            handle_error: |_| {
                rsx! { "Oops, we encountered an error. Please report this to the
developer of this application" }
            },
            ThrowsError {}
        }
    }
}
```

Throwing Errors from Event Handlers

In addition to components, you can throw errors from event handlers. If you throw an error from an event handler, it will bubble up to the nearest error boundary just like a component:

src/error_handling.rs

```
#[component]
fn ThrowsError() -> Element {
    rsx! {
        button {
            onclick: move |_| {
                // Event handlers can return errors just like components
                let number: i32 = "1...234".parse()?;

                tracing::info!("Parsed number: {number}");

                Ok(())
            },
            "Throw error"
        }
    }
}
```

Adding context to errors

You can add additional context to your errors with the [Context](#) trait. Calling `context` on a `Result` will add the context to the error variant of the `Result`:

src/error_handling.rs

```
#[component]
fn ThrowsError() -> Element {
    // You can call the context method on results to add more information to the
    // error
    let number: i32 = use_hook(|| "1.234")
        .parse()
        .context("Failed to parse name")?;

    todo!()
}
```

If you need some custom UI for the error message, you can call `show` on a result to attach an Element to the error variant. The parent error boundary can choose to render this element instead of the default error message:

src/error_handling.rs

```
#[component]
fn Parent() -> Element {
    rsx! {
        ErrorBoundary {
            // The error boundary accepts a closure that will be rendered when
            // an error is thrown in any
            // of the children
            handle_error: |error: ErrorContext| {
                if let Some(error_ui) = error.show() {
                    rsx! {
                        {error_ui}
                    }
                } else {
                    rsx! {
                        div {
                            "Oops, we encountered an error. Please report this
                            to the developer of this application"
                        }
                    }
                }
            },
            ThrowsError {}
        }
    }
}
```

Local Error Handling

If you need more fine-grained control over error states, you can store errors in reactive hooks and use them just like any other value. For example, if you need to show a phone number validation error, you can store the error in a memo and show it below the input field if it is invalid:

src/error_handling.rs

```
#[component]
pub fn PhoneNumberValidation() -> Element {
    let mut phone_number = use_signal(|| String::new());
    let parsed_phone_number = use_memo(move ||
    phone_number().parse:::<PhoneNumber>());

    rsx! {
        input {
            class: "border border-gray-300 rounded-md p-2 mb-4",
            placeholder: "Phone number",
            value: "{phone_number}",
            oninput: move |e| {
                phone_number.set(e.value());
            },
        }

        match parsed_phone_number() {
            Ok(phone_number) => rsx! {
                div {
                    "Parsed phone number: {phone_number}"
                }
            }
        }
    }
}
```

```

    }
  },
  Err(error) => rsx! {
    div {
      "Phone number is invalid: {error}"
    }
  }
}
}
}
}

```

Phone number

Phone number is invalid: failed to parse phone number

The Rules of Hooks

Hooks are a powerful way to manage state in Dioxus, but there are some rules you need to follow to insure they work as expected. Dioxus uses the order you call hooks to differentiate between hooks. Because the order you call hooks matters, you must follow these rules:

1. Hooks may be only used in components or other hooks (we'll get to that later)
2. On every call to the component function
 1. The same hooks must be called
 2. In the same order
3. Hooks name's should start with `use_` so you don't accidentally confuse them with regular functions

These rules mean that there are certain things you can't do with hooks:

No Hooks in Conditionals

src/hooks_bad.rs

```

// ✗ don't call hooks in conditionals!
// We must ensure that the same hooks will be called every time
// But `if` statements only run if the conditional is true!
// So we might violate rule 2.
if you_are_happy && you_know_it {
  let something = use_signal(|| "hands");
  println!("clap your {something}")
}

// ✓ instead, *always* call use_signal
// You can put other stuff in the conditional though
let something = use_signal(|| "hands");
if you_are_happy && you_know_it {
  println!("clap your {something}")
}

```

No Hooks in Closures

src/hooks_bad.rs

```
// ✗ don't call hooks inside closures!
// We can't guarantee that the closure, if used, will be called in the same
order every time
let _a = || {
    let b = use_signal(|| 0);
    b()
};

// ✓ instead, move hook `b` outside
let b = use_signal(|| 0);
let _a = || b();
```

No Hooks in Loops

src/hooks_bad.rs

```
// `names` is a Vec<&str>

// ✗ Do not use hooks in loops!
// In this case, if the length of the Vec changes, we break rule 2
for _name in &names {
    let is_selected = use_signal(|| false);
    println!("selected: {is_selected}");
}

// ✓ Instead, use a hashmap with use_signal
let selection_map = use_signal(HashMap::<&str, bool>::new);

for name in &names {
    let is_selected = selection_map.read()[name];
    println!("selected: {is_selected}");
}
```

Introduction

If you are not familiar with Dioxus itself, check out the [Dioxus guide](#) first.

Whether you are building a website, desktop app, or mobile app, splitting your app's views into "pages" can be an effective method for organization and maintainability.

For this purpose, Dioxus provides a router. Use the `cargo add` command to add the dependency:

```
cargo add dioxus --features router
```

Then, add this to your `Dioxus.toml` (learn more about configuration [here](#)):

```
[web.watcher]
index_on_404 = true
```

This configuration only works when using `dx serve`. If you host your app in a different way (which you most likely do in production), you need to find out how to add a fallback 404 page to your app, and make it a copy of the generated `dist/index.html`.

This will instruct `dx serve` to redirect any unknown route to the index, to then be resolved by the router. The router works on the client. If we connect through the index route (e.g., `localhost:8080`, then click a link to go to `localhost:8080/contact`), the app renders the new route without reloading. However, when we go to a route *before* going to the index (go straight to `localhost:8080/contact`), we are trying to access a static route from the server, but the only static route on our server is the index (because the Dioxus frontend is a Single Page Application) and it will fail unless we redirect all missing routes to the index.

This book is intended to get you up to speed with Dioxus Router. It is split into two sections:

1. The [reference](#) section explains individual features in depth. You can read it from start to finish, or you can read individual chapters in whatever order you want.
2. If you prefer a learning-by-doing approach, you can check out the [example project](#). It guides you through creating a dioxus app, setting up the router, and using some of its functionality.

Please note that this is not the only documentation for the Dioxus Router. You can also check out the [API Docs](#).

Overview

In this guide, you'll learn to effectively use Dioxus Router whether you're building a small todo app or the next FAANG company. We will create a small website with a blog, homepage, and more!

To follow along with the router example, you'll need a working Dioxus app. Check out the [Dioxus book](#) to get started.

Make sure to add Dioxus Router as a dependency, as explained in the [introduction](#).

You'll learn how to

- Create routes and render "pages".
- Utilize nested routes, create a navigation bar, and render content for a set of routes.
- Parse URL parameters to dynamically display content.
- Redirect visitors to different routes.

Disclaimer

The example will only display the features of Dioxus Router. It will not include any actual functionality. To keep things simple we will only be using a single file, this is not the recommended way of doing things with a real application.

You can find the complete application in the [full code](#) chapter.

Creating Our First Route

In this chapter, we will start utilizing Dioxus Router and add a homepage and a 404 page to our project.

Fundamentals

The core of the Dioxus Router is the [Routable](#) macro and the [Router](#) component.

Routable is a trait for anything that can:

- Be parsed from a URL
- Be turned into a URL
- Be rendered as to a Element

Let's create a new router. First, we need an actual page to route to! Let's add a homepage component:

src/first_route.rs

```
#[component]
fn Home() -> Element {
    rsx! { h1 { "Welcome to the Dioxus Blog!" } }
}
```

Creating Routes

We want to use Dioxus Router to separate our application into different "pages". Dioxus Router will then determine which page to render based on the URL path.

To start using Dioxus Router, we need to use the [Routable](#) macro.

The [Routable](#) macro takes an enum with all of the possible routes in our application. Each variant of the enum represents a route and must be annotated with the `#[route(path)]` attribute.

src/first_route.rs

```
#![allow(non_snake_case)]
use dioxus::prelude::*;
use dioxus_router::prelude::*;

/// An enum of all of the possible routes in the app.
#[derive(Routable, Clone)]
enum Route {
    // The home page is at the / route
    #[route("/")]
    Home {},
}
```

The [Router](#) component will provide a router context for all the inner components and hooks to use. You usually will want to place this at the top of your components tree.

src/first_route.rs

```
fn App() -> Element {
    rsx! { Router::<Route> {} }
}
```

If you head to your application's browser tab, you should now see the text `Welcome to Dioxus Blog!` when on the root URL (`http://localhost:8080/`). If you enter a different path for the URL, nothing should be displayed.

This is because we told Dioxus Router to render the `Home` component only when the URL path is `/`.

Fallback Route

In our example, when a route doesn't exist Dioxus Router doesn't render anything. Many sites also have a "404" page when a path does not exist. Let's add one to our site.

First, we create a new `PageNotFound` component.

`src/catch_all.rs`

```
#[component]
fn PageNotFound(route: Vec<String>) -> Element {
    rsx! {
        h1 { "Page not found" }
        p { "We are terribly sorry, but the page you requested doesn't exist." }
        pre { color: "red", "log:\nattempted to navigate to: {route:?}" }
    }
}
```

Next, register the route in the `Route` enum to match if all other routes fail.

`src/catch_all.rs`

```
#[derive(Routable, Clone)]
enum Route {
    #[route("/")]
    Home {},
    // PageNotFound is a catch all route that will match any route and placing
    // the matched segments in the route field
    #[route("/:..route")]
    PageNotFound { route: Vec<String> },
}
```

Now when you go to a route that doesn't exist, you should see the page not found text.

Conclusion

In this chapter, we learned how to create a route and tell Dioxus Router what component to render when the URL path is `/`. We also created a 404 page to handle when a route doesn't exist. Next, we'll create the blog portion of our site. We will utilize nested routes and URL parameters.

Building a Nest

In this chapter, we will begin to build the blog portion of our site which will include links, nested routes, and route parameters.

Site Navigation

Our site visitors won't know all the available pages and blogs on our site so we should provide a navigation bar for them. Our navbar will be a list of links going between our pages.

We want our navbar component to be rendered on several different pages on our site. Instead of duplicating the code, we can create a component that wraps all children routes. This is called a layout component. To tell the router where to render the child routes, we use the [Outlet](#) component.

Let's create a new NavBar component:

src/nested_routes.rs

```
#[component]
fn NavBar() -> Element {
    rsx! {
        nav {
            ul { li { "links" } }
        }
        // The Outlet component will render child routes (In this case just the
        // Home component) inside the Outlet component
        Outlet::<Route> {}
    }
}
```

Next, let's add our NavBar component as a layout to our Route enum:

src/nested_routes.rs

```
#[derive(Routable, Clone)]
#[rustfmt::skip]
enum Route {
    // All routes under the NavBar layout will be rendered inside of the NavBar
    // Outlet
    #[layout(NavBar)]
    #[route("/")]
    Home {},
    #[end_layout]
    #[route("/:..route")]
    PageNotFound { route: Vec<String> },
}
```

To add links to our NavBar, we could always use an HTML anchor element but that has two issues:

1. It causes a full-page reload
2. We can accidentally link to a page that doesn't exist

Instead, we want to use the [Link](#) component provided by Dioxus Router.

The [Link](#) is similar to a regular `<a>` tag. It takes a target and children.

Unlike a regular `<a>` tag, we can pass in our Route enum as the target. Because we annotated our routes with the `#[route(path)]` attribute, the [Link](#) will know how to generate the correct URL. If we use the Route enum, the rust compiler will prevent us from linking to a page that doesn't exist.

Let's add our links:

src/links.rs

```
#[component]
fn NavBar() -> Element {
```

```

    rsx! {
      nav {
        ul {
          li {
            Link { to: Route::Home {}, "Home" }
          }
        }
      }
      Outlet::<Route> {}
    }
  }
}

```

Using this method, the [Link](#) component only works for links within our application. To learn more about navigation targets see [here](#).

Now you should see a list of links near the top of your page. Click on one and you should seamlessly travel between pages.

URL Parameters and Nested Routes

Many websites such as GitHub put parameters in their URL. For example, <https://github.com/DioxusLabs> utilizes the text after the domain to dynamically search and display content about an organization.

We want to store our blogs in a database and load them as needed. We also want our users to be able to send people a link to a specific blog post. Instead of listing all of the blog titles at compile time, we can make a dynamic route.

We could utilize a search page that loads a blog when clicked but then our users won't be able to share our blogs easily. This is where URL parameters come in.

The path to our blog will look like `/blog/myBlogPage`, `myBlogPage` being the URL parameter.

First, let's create a layout component (similar to the navbar) that wraps the blog content. This allows us to add a heading that tells the user they are on the blog.

`src/dynamic_route.rs`

```

#[component]
fn Blog() -> Element {
  rsx! {
    h1 { "Blog" }
    Outlet::<Route> {}
  }
}

```

Now we'll create another index component, that'll be displayed when no blog post is selected:

`src/dynamic_route.rs`

```

#[component]
fn BlogList() -> Element {
  rsx! {
    h2 { "Choose a post" }
    ul {
      li {
        Link {

```

```

        to: Route::BlogPost {
            name: "Blog post 1".into(),
        },
        "Read the first blog post"
    }
}
li {
    Link {
        to: Route::BlogPost {
            name: "Blog post 2".into(),
        },
        "Read the second blog post"
    }
}
}
}
}
```

We also need to create a component that displays an actual blog post. This component will accept the URL parameters as props:

src/dynamic_route.rs

```
// The name prop comes from the /:name route segment
#[component]
fn BlogPost(name: String) -> Element {
  rsx! { h2 { "Blog Post: {name}" } }
}
```

Finally, let's tell our router about those components:

src/dynamic_route.rs

```
#[derive(Routable, Clone)]
#[rustfmt::skip]
enum Route {
    #[layout(NavBar)]
    #[route("/")]
    Home {},
    #[nest("/blog")]
    #[layout(Blog)]
    #[route("/")]
    BlogList {},
    #[route("/post/:name")]
    BlogPost { name: String },
    #[end_layout]
    #[end_nest]
    #[end_layout]
    #[route("/:..route")]
    PageNotFound {
        route: Vec<String>,
    },
}
```

That's it! If you head to `/blog/1` you should see our sample post.

Conclusion

In this chapter, we utilized Dioxus Router's Link, and Route Parameter functionality to build the blog portion of our application. In the next chapter, we will go over how navigation targets (like the one we passed to our links) work.

Navigation Targets

In the previous chapter, we learned how to create links to pages within our app. We told them where to go using the `target` property. This property takes something that can be converted to a [`NavigationTarget`](#).

What is a navigation target?

A [`NavigationTarget`](#) is similar to the `href` of an HTML anchor element. It tells the router where to navigate to. The Dioxus Router knows two kinds of navigation targets:

- [`Internal`](#): We used internal links in the previous chapter. It's a link to a page within our app represented as a Route enum.
- [`External`](#): This works exactly like an HTML anchors' `href`. Don't use this for in-app navigation as it will trigger a page reload by the browser.

External navigation

If we need a link to an external page we can do it like this:

src/external_link.rs

```
fn GoToDioxus() -> Element {
  rsx! {
    Link { to: "https://dioxuslabs.com", "ExternalTarget target" }
  }
}
```

Redirection Perfection

You're well on your way to becoming a routing master!

In this chapter, we will cover creating redirects

Creating Redirects

A redirect is very simple. When dioxus encounters a redirect while finding out what components to render, it will redirect the user to the target of the redirect.

As a simple example, let's say you want user to still land on your blog, even if they used the path `/myblog` or `/myblog/:name`.

Redirects are special attributes in the router enum that accept a route and a closure with the route parameters. The closure should return a route to redirect to.

Let's add a redirect to our router enum:

src/full_example.rs

```
#[derive(Routable, Clone)]
#[rustfmt::skip]
enum Route {
    #[layout(NavBar)]
    #[route("/")]
    Home {},
    #[nest("/blog")]
    #[layout(Blog)]
    #[route("/")]
    BlogList {},
    #[route("/post/:name")]
    BlogPost { name: String },
    #[end_layout]
    #[end_nest]
    #[end_layout]
    #[nest("/myblog")]
    #[redirect("/", || Route::BlogList {})]
    #[redirect("/:name", |name: String| Route::BlogPost { name })]
    #[end_nest]
    #[route("/:..route")]
    PageNotFound {
        route: Vec<String>,
    },
}
```

That's it! Now your users will be redirected to the blog.

Conclusion

Well done! You've completed the Dioxus Router guide. You've built a small application and learned about the many things you can do with Dioxus Router. To continue your journey, you attempt a challenge listed below, look at the [router examples](#), or the [API reference](#).

Challenges

- Organize your components into separate files for better maintainability.
- Give your app some style if you haven't already.
- Build an about page so your visitors know who you are.
- Add a user system that uses URL parameters.
- Create a simple admin system to create, delete, and edit blogs.
- If you want to go to the max, hook up your application to a rest API and database.

Full Code

src/full_example.rs

```
#![allow(non_snake_case)]

use dioxus::prelude::*;
use dioxus_router::prelude::*;

// ANCHOR: router
#[derive(Routable, Clone)]
```

```

#[rustfmt::skip]
enum Route {
    #[layout(NavBar)]
    #[route("/")]
    Home {},
    #[nest("/blog")]
    #[layout(Blog)]
    #[route("/")]
    BlogList {},
    #[route("/post/:name")]
    BlogPost { name: String },
    #[end_layout]
    #[end_nest]
    #[end_layout]
    #[nest("/myblog")]
    #[redirect("/", || Route::BlogList {})]
    #[redirect("/:name", |name: String| Route::BlogPost { name })]
    #[end_nest]
    #[route("/:..route")]
    PageNotFound {
        route: Vec<String>,
    },
}
// ANCHOR_END: router

pub fn App() -> Element {
    rsx! { Router::<Route> {} }
}

#[component]
fn NavBar() -> Element {
    rsx! {
        nav {
            ul {
                li {
                    Link { to: Route::Home {}, "Home" }
                }
                li {
                    Link { to: Route::BlogList {}, "Blog" }
                }
            }
        }
        Outlet::<Route> {}
    }
}

#[component]
fn Home() -> Element {
    rsx! { h1 { "Welcome to the Dioxus Blog!" } }
}

#[component]
fn Blog() -> Element {
    rsx! {
        h1 { "Blog" }
        Outlet::<Route> {}
    }
}

#[component]
fn BlogList() -> Element {
    rsx! {
        h2 { "Choose a post" }
        ul {

```

```

        li {
            Link {
                to: Route::BlogPost {
                    name: "Blog post 1".into(),
                },
                "Read the first blog post"
            }
        }
        li {
            Link {
                to: Route::BlogPost {
                    name: "Blog post 2".into(),
                },
                "Read the second blog post"
            }
        }
    }
}

#[component]
fn BlogPost(name: String) -> Element {
    rsx! { h2 { "Blog Post: {name}" } }
}

#[component]
fn PageNotFound(route: Vec<String>) -> Element {
    rsx! {
        h1 { "Page not found" }
        p { "We are terribly sorry, but the page you requested doesn't exist." }
        pre { color: "red", "log:\nattempted to navigate to: {route:?}" }
    }
}

```

Adding the router to your application

In this chapter, we will learn how to add the router to our app. By itself, this is not very useful. However, it is a prerequisite for all the functionality described in the other chapters.

Make sure you added the `dioxus-router` dependency as explained in the [introduction](#).

In most cases, we want to add the router to the root component of our app. This way, we can ensure that we have access to all its functionality everywhere.

First, we define the router with the router macro:

src/first_route.rs

```

#![allow(non_snake_case)]
use dioxus::prelude::*;
use dioxus_router::prelude::*;

/// An enum of all of the possible routes in the app.
#[derive(Routable, Clone)]
enum Route {
    // The home page is at the / route
    #[route("/")]
    Home {},
}

```

Then we render the router with the [Router] component.

src/first_route.rs

```
fn App() -> Element {  
    rsx! { Router::<Route> {} }  
}
```

Defining Routes

When creating a [Routable] enum, we can define routes for our application using the route("path") attribute.

Route Segments

Each route is made up of segments. Most segments are separated by / characters in the path.

There are four fundamental types of segments:

1. Static segments are fixed strings that must be present in the path.
2. Dynamic segments are types that can be parsed from a segment.
3. Catch-all segments are types that can be parsed from multiple segments.
4. Query segments are types that can be parsed from the query string.

Routes are matched:

- First, from most specific to least specific (Static then Dynamic then Catch All) (Query is always matched)
- Then, if multiple routes match the same path, the order in which they are defined in the enum is followed.

Static segments

Fixed routes match a specific path. For example, the route #[route("/about")] will match the path /about.

src/static_segments.rs

```
#[derive(Routable, Clone)]  
#[rustfmt::skip]  
enum Route {  
    // Routes always start with a slash  
    #[route("/")]  
    Home {},  
    // You can have multiple segments in a route  
    #[route("/hello/world")]  
    HelloWorld {},  
}  
  
#[component]  
fn Home() -> Element {  
    todo!()  
}  
  
#[component]
```

```
fn HelloWorld() -> Element {
    todo!()
}
```

Dynamic Segments

Dynamic segments are in the form of `:name` where `name` is the name of the field in the route variant. If the segment is parsed successfully then the route matches, otherwise the matching continues.

The segment can be of any type that implements `FromStr`.

src/dynamic_segments.rs

```
#[derive(Routable, Clone)]
#[rustfmt::skip]
enum Route {
    // segments that start with : are dynamic segments
    #[route("/post/:name")]
    BlogPost {
        // You must include dynamic segments in child variants
        name: String,
    },
    #[route("/document/:id")]
    Document {
        // You can use any type that implements FromStr
        // If the segment can't be parsed, the route will not match
        id: usize,
    },
}

// Components must contain the same dynamic segments as their corresponding
// variant
#[component]
fn BlogPost(name: String) -> Element {
    todo!()
}

#[component]
fn Document(id: usize) -> Element {
    todo!()
}
```

Catch All Segments

Catch All segments are in the form of `:. . name` where `name` is the name of the field in the route variant. If the segments are parsed successfully then the route matches, otherwise the matching continues.

The segment can be of any type that implements `FromSegments`. (`Vec` implements this by default)

Catch All segments must be the *last route segment* in the path (query segments are not counted) and cannot be included in nests.

src/catch_all_segments.rs

```
#[derive(Routable, Clone)]
#[rustfmt::skip]
enum Route {
    // segments that start with :.. are catch all segments
    #[route("/blog/:..segments")]
    BlogPost {
        // You must include catch all segment in child variants
        segments: Vec<String>,
    },
}

// Components must contain the same catch all segments as their corresponding
variant
#[component]
fn BlogPost(segments: Vec<String>) -> Element {
    todo!()
}
```

Query Segments

Query segments are in the form of `?:name&:othername` where `name` and `othername` are the names of fields in the route variant.

Unlike [Dynamic Segments](#) and [Catch All Segments](#), parsing a Query segment must not fail.

The segment can be of any type that implements `FromQueryArgument`.

Query segments must be the *after all route segments* and cannot be included in nests.

src/query_segments.rs

```
#[derive(Routable, Clone)]
#[rustfmt::skip]
enum Route {
    // segments that start with ?: are query segments
    #[route("/blog?:name&:surname")]
    BlogPost {
        // You must include query segments in child variants
        name: String,
        surname: String,
    },
}

#[component]
fn BlogPost(name: String, surname: String) -> Element {
    rsx! {
        div { "This is your blogpost with a query segment:" }
        div { "Name: {name}" }
        div { "Surname: {surname}" }
    }
}

fn App() -> Element {
    rsx! { Router::<Route> {} }
}

fn main() {}
```

Nested Routes

When developing bigger applications we often want to nest routes within each other. As an example, we might want to organize a settings menu using this pattern:

```
└ Settings
  ├── General Settings (displayed when opening the settings)
  ├── Change Password
  └ Privacy Settings
```

We might want to map this structure to these paths and components:

```
/settings          -> Settings { GeneralSettings }
/settings/password -> Settings { PWSettings }
/settings/privacy  -> Settings { PrivacySettings }
```

Nested routes allow us to do this without repeating `/settings` in every route.

Nesting

To nest routes, we use the `#[nest("path")]` and `#[end_nest]` attributes.

The path in nest must not:

1. Contain a [Catch All Segment](#)
2. Contain a [Query Segment](#)

If you define a dynamic segment in a nest, it will be available to all child routes and layouts.

To finish a nest, we use the `#[end_nest]` attribute or the end of the enum.

src/nest.rs

```
#[derive(Routable, Clone)]
// Skipping formatting allows you to indent nests
#[rustfmt::skip]
enum Route {
    // Start the /blog nest
    #[nest("/blog")]
    // You can nest as many times as you want
    #[nest("/:id")]
    #[route("/post")]
    PostId {
        // You must include parent dynamic segments in child variants
        id: usize,
    },
    // End nests manually with #[end_nest]
    #[end_nest]
    #[route("/:id")]
    // The absolute route of BlogPost is /blog/:name
    BlogPost {
        id: usize,
    },
    // Or nests are ended automatically at the end of the enum
}

#[component]
fn BlogPost(id: usize) -> Element {
    todo!()
}
```

```
#[component]
fn PostId(id: usize) -> Element {
    todo!()
}
```

Layouts

Layouts allow you to wrap all child routes in a component. This can be useful when creating something like a header that will be used in many different routes.

[Outlet](#) tells the router where to render content in layouts. In the following example, the Index will be rendered within the [Outlet](#).

This page is built with the Dioxus. It uses Layouts in several different places. Here is an outline of how layouts are used on the current page. Hover over different layouts to see what elements they are on the page.

```
#[derive(Clone, Routable, PartialEq, Eq, Serialize, Deserialize)]
#[rustfmt::skip]
pub enum Route {
    #[layout(HeaderFooter)]
    // ... other routes
    #[layout(DocsSidebars)]
    #[route("/learn")]
    Docs {},
}
```

Here is a more complete example of a layout wrapping the body of a page.

src/outlet.rs

```
#[derive(Routable, Clone)]
#[rustfmt::skip]
enum Route {
    #[layout(Wrapper)]
    #[route("/")]
    Index {},
}

#[component]
fn Wrapper() -> Element {
    rsx! {
        header { "header" }
        // The index route will be rendered here
        Outlet::<Route> {}
        footer { "footer" }
    }
}

#[component]
fn Index() -> Element {
    rsx! { h1 { "Index" } }
}
```

The example above will output the following HTML (line breaks added for readability):

```
<header>header</header>
<h1>Index</h1>
```

```
<footer>footer</footer>
```

Layouts with dynamic segments

You can combine layouts with [nested routes](#) to create dynamic layouts with content that changes based on the current route.

Just like routes, layouts components must accept a prop for each dynamic segment in the route. For example, if you have a route with a dynamic segment like `/:name`, your layout component must accept a `name` prop:

src/outlet.rs

```
#[derive(Routable, Clone)]
#[rustfmt::skip]
enum Route {
    #[nest("/:name")]
    #[layout(Wrapper)]
    #[route("/")]
    Index {
        name: String,
    },
}

#[component]
fn Wrapper(name: String) -> Element {
    rsx! {
        header { "Welcome {name}!" }
        // The index route will be rendered here
        Outlet::<Route> {}
        footer { "footer" }
    }
}

#[component]
fn Index(name: String) -> Element {
    rsx! { h1 { "This is a homepage for {name}" } }
}
```

Or to get the full route, you can use the `use_route` hook.

src/outlet.rs

```
#[derive(Routable, Clone)]
#[rustfmt::skip]
enum Route {
    #[layout(Wrapper)]
    #[route("/:name")]
    Index {
        name: String,
    },
}

#[component]
fn Wrapper() -> Element {
    let full_route = use_route::<Route>();
    rsx! {
        header { "Welcome to {full_route}!" }
        // The index route will be rendered here
        Outlet::<Route> {}
    }
}
```

```

    footer { "footer" }
  }
}

#[component]
fn Index(name: String) -> Element {
  rsx! { h1 { "This is a homepage for {name}" } }
}

```

Links & Navigation

When we split our app into pages, we need to provide our users with a way to navigate between them. On regular web pages, we'd use an anchor element for that, like this:

```
<a href="/other">Link to an other page</a>
```

However, we cannot do that when using the router for three reasons:

1. Anchor tags make the browser load a new page from the server. This takes a lot of time, and it is much faster to let the router handle the navigation client-side.
2. Navigation using anchor tags only works when the app is running inside a browser. This means we cannot use them inside apps using Dioxus Desktop.
3. Anchor tags cannot check if the target page exists. This means we cannot prevent accidentally linking to non-existent pages.

To solve these problems, the router provides us with a [`Link`] component we can use like this:

src/links.rs

```

#[component]
fn NavBar() -> Element {
  rsx! {
    nav {
      ul {
        li {
          Link { to: Route::Home {}, "Home" }
        }
      }
    }
    Outlet::<Route> {}
  }
}

```

The `target` in the example above is similar to the `href` of a regular anchor element. However, it tells the router more about what kind of navigation it should perform. It accepts something that can be converted into a [`NavigationTarget`]:

- The example uses a `Internal` route. This is the most common type of navigation. It tells the router to navigate to a page within our app by passing a variant of a [`Routeable`] enum. This type of navigation can never fail if the link component is used inside a router component.
- [`External`] allows us to navigate to URLs outside of our app. This is useful for links to external websites. `NavigationTarget::External` accepts an URL to navigate to. This type of navigation can fail if the URL is invalid.

The `[ Link ]` accepts several props that modify its behavior. See the API docs for more details.

Programmatic Navigation

Sometimes we want our application to navigate to another page without having the user click on a link. This is called programmatic navigation.

Using a Navigator

We can get a navigator with the `navigator` function which returns a `Navigator`.

We can use the `Navigator` to trigger four different kinds of navigation:

- `push` will navigate to the target. It works like a regular anchor tag.
- `replace` works like `push`, except that it replaces the current history entry instead of adding a new one. This means the prior page cannot be restored with the browser's back button.
- `Go back` works like the browser's back button.
- `Go forward` works like the browser's forward button.

src/navigator.rs

```
#[component]
fn Home() -> Element {
    let nav = navigator();

    // push
    nav.push(Route::PageNotFound { route: vec![] });

    // replace
    nav.replace(Route::Home {});

    // go back
    nav.go_back();

    // go forward
    nav.go_forward();

    rsx! { h1 { "Welcome to the Dioxus Blog!" } }
}
```

You might have noticed that, like `Link`, the `Navigator`'s `push` and `replace` functions take a `NavigationTarget`. This means we can use either `Internal`, or `External` targets.

External Navigation Targets

Unlike a `Link`, the `Navigator` cannot rely on the browser (or webview) to handle navigation to external targets via a generated anchor element.

This means, that under certain conditions, navigation to external targets can fail.

History Providers

[[HistoryProvider](#)] s are used by the router to keep track of the navigation history and update any external state (e.g. the browser's URL).

The router provides two [[HistoryProvider](#)] s, but you can also create your own. The two default implementations are:

- The [[MemoryHistory](#)] is a custom implementation that works in memory.
- The [[LiveviewHistory](#)] is a custom implementation that works with the liveview renderer.
- The [[WebHistory](#)] integrates with the browser's URL.

By default, the router uses the [[MemoryHistory](#)] . It might be changed to use [[WebHistory](#)] when the `web` feature is active, but that is not guaranteed.

You can override the default history:

src/history_provider.rs

```
#[component]
fn App() -> Element {
  rsx! {
    Router::<Route> { config: || RouterConfig::default() }
  }
}
```

History Buttons

Some platforms, like web browsers, provide users with an easy way to navigate through an app's history. They have UI elements or integrate with the OS.

However, native platforms usually don't provide such amenities, which means that apps wanting users to have access to them, need to implement them. For this reason, the router comes with two components, which emulate a browser's back and forward buttons:

- [GoBackButton](#)
- [GoForwardButton](#)

If you want to navigate through the history programmatically, take a look at [programmatic navigation](#).

src/history_buttons.rs

```
fn HistoryNavigation() -> Element {
  rsx! {
    GoBackButton { "Back to the Past" }
    GoForwardButton { "Back to the Future" }
  }
}
```

As you might know, browsers usually disable the back and forward buttons if there is no history to navigate to. The router's history buttons try to do that too, but depending on the [[history provider](#)] that might not be possible.

Importantly, neither `WebHistory` supports that feature. This is due to limitations of the browser History API.

However, in both cases, the router will just ignore button presses, if there is no history to navigate to.

Also, when using `WebHistory`, the history buttons might navigate a user to a history entry outside your app.

Routing Update Callback

In some cases, we might want to run custom code when the current route changes. For this reason, the `RouterConfig` exposes an `on_update` field.

How does the callback behave?

The `on_update` is called whenever the current routing information changes. It is called after the router updated its internal state, but before dependent components and hooks are updated.

If the callback returns a `NavigationTarget`, the router will replace the current location with the specified target. It will not call the `on_update` again.

If at any point the router encounters a navigation failure, it will go to the appropriate state without calling the `on_update`. It doesn't matter if the invalid target initiated the navigation, was found as a redirect target, or was returned by the `on_update` itself.

Code Example

src/routing_update.rs

```
#[derive(Routable, Clone, PartialEq)]
enum Route {
    #[route("/")]
    Index {},
    #[route("/home")]
    Home {},
}

#[component]
fn Home() -> Element {
    rsx! { p { "Home" } }
}

#[component]
fn Index() -> Element {
    rsx! { p { "Index" } }
}

fn app() -> Element {
    rsx! {
        Router::<Route> {
            config: || {
                RouterConfig::default()
                    .on_update(|state| {
                        (state.current() == Route::Index {})
                    })
            }
        }
    }
}
```

```

        .then_some(NavigationTarget::Internal(Route::Home
    {}))
    })
  }
}
}

```

Assets

Assets are files that are included in the final build of the application. They can be images, fonts, stylesheets, or any other file that is not a source file. Dioxus includes first class support for assets, and provides a simple way to include them in your application and automatically optimize them for production.

Assets in dioxus are also compatible with libraries! If you are building a library, you can include assets in your library and they will be automatically included in the final build of any application that uses your library.

Including images

To include an asset in your application, you can simply wrap the path to the asset in the `asset!` macro. For example, to include an image in your application, you can use the following code:

src/assets.rs

```

use dioxus::prelude::*;

fn App() -> Element {
    // You can link to assets that are relative to the package root or even link
    // to an asset from a url
    // These assets will automatically be picked up by the dioxus cli,
    // optimized, and bundled with your final applications
    const ASSET: Asset = asset!("/assets/static/ferrous_wave.png");

    rsx! {
        img { src: "{ASSET}" }
    }
}

```

You can also optimize, resize, and preload images using the `asset!` macro. Choosing an optimized file type (like Avif) and a reasonable quality setting can significantly reduce the size of your images which helps your application load faster. For example, you can use the following code to include an optimized image in your application:

src/assets.rs

```

pub const ENUM_ROUTER_IMG: Asset = asset!(
    "/assets/static/enum_router.png",
    // You can pass a second argument to the asset macro to set up options for
    // the asset
    ImageAssetOptions::new()
    // You can set the image size in pixels at compile time to send the
    // smallest possible image to the client
    .with_size(ImageSize::Manual {
        width: 52,
    })
);

```

```

        height: 52
    })
    // You can also convert the image to a web friendly format at compile
time. This can make your images significantly smaller
    .with_format(ImageFormat::Avif)
);

fn EnumRouter() -> Element {
    rsx! {
        img { src: "{ENUM_ROUTER_IMG}" }
    }
}

```

Including arbitrary files

In dioxus desktop, you may want to include a file with data for your application. If you don't set any options for your asset and the file extension is not recognized, the asset will be copied without any changes. For example, you can use the following code to include a binary file in your application:

src/assets.rs

```

// You can also collect arbitrary files. Relative paths are resolved relative to
the package root
const PATH_TO_BUNDLED_CARGO_TOML: Asset = asset!("/Cargo.toml");

```

These files will be automatically included in the final build of your application, and you can use them in your application as you would any other file.

Including stylesheets

You can include stylesheets in your application using the `asset!` macro. Stylesheets will automatically be minified as they are bundled to speed up load times. For example, you can use the following code to include a stylesheet in your application:

src/assets.rs

```

// You can also bundle stylesheets with your application
// Any files that end with .css will be minified and bundled with your
application even if you don't explicitly include them in your <head>
const _: Asset = asset!("/assets/tailwind.css");

```

The [tailwind guide](#) has more information on how to use tailwind with dioxus.

Conclusion

Dioxus provides first class support for assets, and makes it easy to include them in your application. You can include images, arbitrary files, and stylesheets in your application, and dioxus will automatically optimize them for production. This makes it easy to include assets in your application and ensure that they are optimized for production.

You can read more about assets and all the options available to optimize your assets in the [manganis documentation](#).

Web

To run on the Web, your app must be compiled to WebAssembly and depend on the `dioxus` and `dioxus-web` crates.

A build of Dioxus for the web will be roughly equivalent to the size of a React build (70kb vs 65kb) but it will load significantly faster because [WebAssembly can be compiled as it is streamed](#).

Examples:

- [TodoMVC](#)
- [Tailwind App](#)



Note: Because of the limitations of Wasm, [not every crate will work](#) with your web apps, so you'll need to make sure that your crates work without native system calls (timers, IO, etc).

Support

The Web is the best-supported target platform for Dioxus.

- Because your app will be compiled to WASM you have access to browser APIs through [wasm-bindgen](#).
- Dioxus provides hydration to resume apps that are rendered on the server. See the [fullstack](#) reference for more information.

Running Javascript

Dioxus provides some ergonomic wrappers over the browser API, but in some cases you may need to access parts of the browser API Dioxus does not expose.

For these cases, Dioxus web exposes the `use_eval` hook that allows you to run raw Javascript in the webview:

src/eval.rs

```
use dioxus::prelude::*;

fn main() {
    launch(app);
}

fn app() -> Element {
    let future = use_resource(move || async move {
        // You can create as many eval instances as you want
        let mut eval = document::eval(
            r#"
            // You can send messages from JavaScript to Rust with the
dioxus.send function
            dioxus.send("Hi from JS!");
            // You can receive messages from Rust to JavaScript with the
dioxus.recv function
            let msg = await dioxus.recv();
            console.log(msg);
            "#,
        );

        // You can send messages to JavaScript with the send method
        eval.send("Hi from Rust!").unwrap();

        // You can receive any message from JavaScript with the recv method
        eval.recv:::<String>().await.unwrap()
    });

    match future.read_unchecked().as_ref() {
        Some(v) => rsx! {
            p { "{v}" }
        },
        _ => rsx! {
            p { "hello" }
        },
    }
}
```

If you are targeting web, but don't plan on targeting any other Dioxus renderer you can also use the generated wrappers in the [web-sys](#) and [gloo](#) crates.

Customizing Index Template

Dioxus supports providing custom index.html templates. The index.html must include a `div` with the id `main` to be used. Hot Reload is still supported. An example is provided in the [PWA-Example](#).

Desktop

This guide will cover concepts specific to the Dioxus desktop renderer.

Apps built with Dioxus desktop use the system WebView to render the page. This makes the final size of application much smaller than other WebView renderers (typically under 5MB).

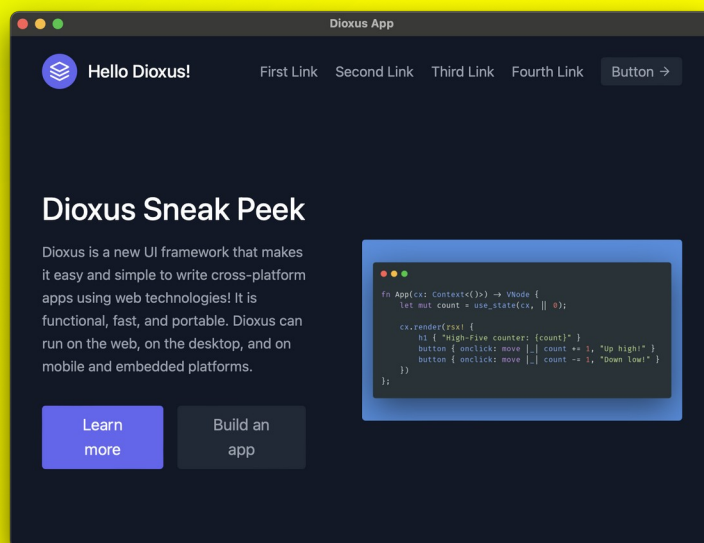
Although desktop apps are rendered in a WebView, your Rust code runs natively. This means that browser APIs are *not* available, so rendering WebGL, Canvas, etc is not as easy as the Web. However, native system APIs *are* accessible, so streaming, WebSockets, filesystem, etc are all easily accessible through system APIs.

Dioxus desktop is built on top of [wry](#), a Rust library for creating desktop applications with a WebView.

In the future, we plan to move to a custom web renderer-based DOM renderer with WGPU integrations ([Blitz](#)).

Examples

- [File Explorer](#)
- [Tailwind App](#)



Running Javascript

Dioxus provides some ergonomic wrappers over the browser API, but in some cases you may need to access parts of the browser API Dioxus does not expose.

For these cases, Dioxus desktop exposes the `use_eval` hook that allows you to run raw Javascript in the webview:

`src/eval.rs`

```

use dioxus::prelude::*;

fn main() {
    launch(app);
}

fn app() -> Element {
    let future = use_resource(move || async move {
        // You can create as many eval instances as you want
        let mut eval = document::eval(
            r#"
            // You can send messages from JavaScript to Rust with the
dioxus.send function
            dioxus.send("Hi from JS!");
            // You can receive messages from Rust to JavaScript with the
dioxus.recv function
            let msg = await dioxus.recv();
            console.log(msg);
            "#,
        );

        // You can send messages to JavaScript with the send method
        eval.send("Hi from Rust!").unwrap();

        // You can receive any message from JavaScript with the recv method
        eval.recv:::<String>().await.unwrap()
    });

    match future.read_unchecked().as_ref() {
        Some(v) => rsx! {
            p { "{v}" }
        },
        _ => rsx! {
            p { "hello" }
        },
    }
}

```

Custom Assets

You can link to local assets in dioxus desktop instead of using a url:

src/custom_assets.rs

```

use dioxus::prelude::*;

fn main() {
    launch(app);
}

fn app() -> Element {
    rsx! {
        div {
            img { src: asset!("/assets/static/scanner.png") }
        }
    }
}

```

You can read more about assets in the [assets](#) reference.

Integrating with Wry

In cases where you need more low level control over your window, you can use wry APIs exposed through the [Desktop Config](#) and the [use_window hook](#)

Mobile App

Build a mobile app with Dioxus!

Support

The Rust ecosystem for mobile continues to mature, with Dioxus offering strong support for mobile applications. Mobile is a first-class target for Dioxus apps, with a robust WebView implementation that supports CSS animations and transparency effects.

Mobile apps are rendered with either the platform's WebView or experimentally with WGPU. While native Android animations and widgets aren't currently supported, CSS-based animations and styling provide a powerful alternative.

Mobile support is well-suited for most application types, from business tools to consumer apps, making it an excellent choice for teams looking to build cross-platform applications with a single codebase.

Getting Set up

Android

Android devices run a different executable architecture than desktop and web. We need to install these toolchains to build Dioxus apps for Android.

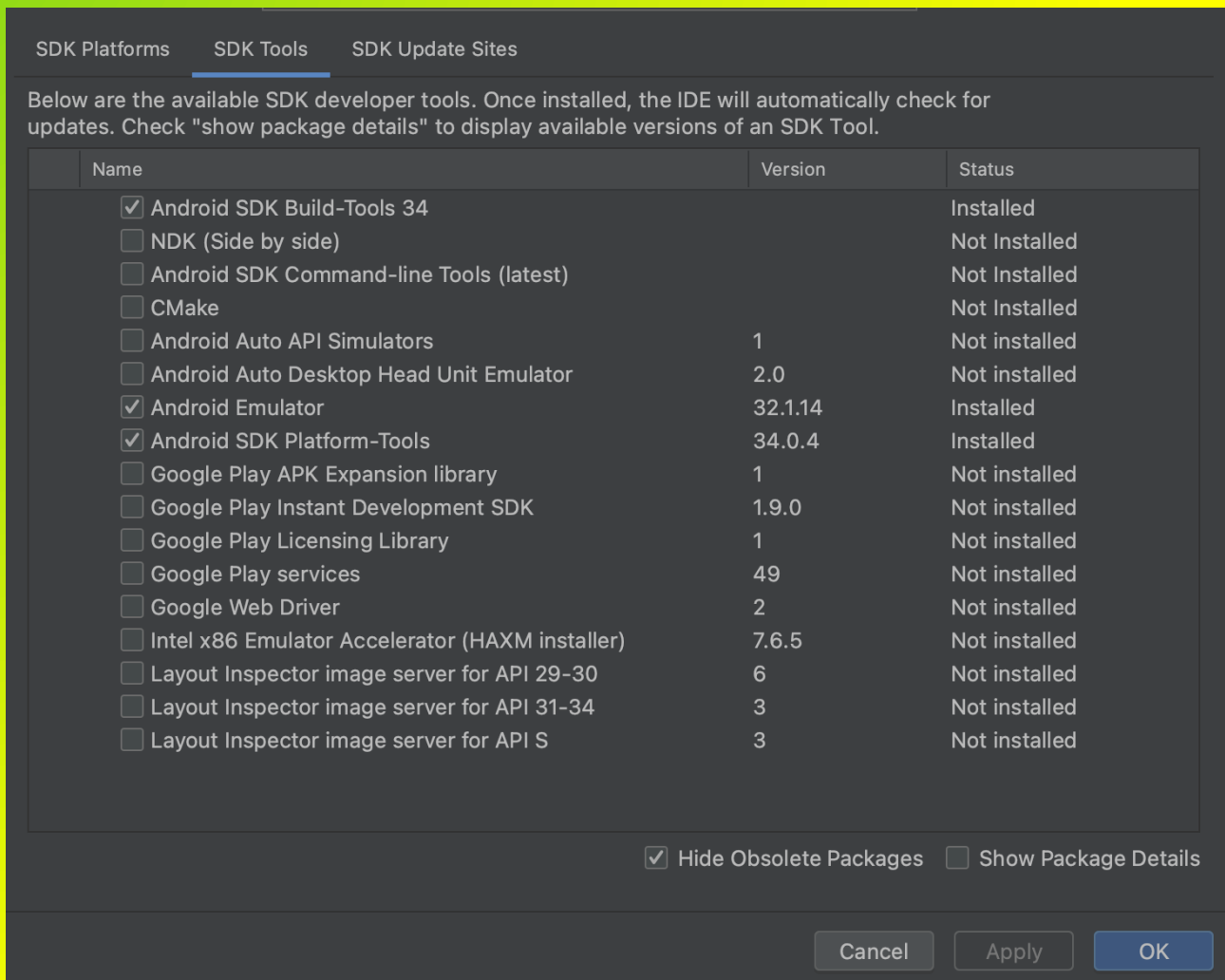
First, install the Rust Android targets:

```
rustup target add aarch64-linux-android armv7-linux-androideabi i686-linux-android x86_64-linux-android
```

To develop on Android, you will need to [install Android Studio](#).

Once you have installed Android Studio, you will need to install the Android SDK and NDK:

1. Create a blank Android project
2. Select **Tools > SDK manager**
3. Navigate to the **SDK tools** window:



Then select:

- The SDK
- The SDK Command line tools
- The NDK (side by side)
- CMAKE

1. Select **apply** and follow the prompts

More details that could be useful for debugging any errors you encounter are available [in the official android docs](#)

Next set the Java, Android, NDK, and PATH variables:

Mac:

```
export JAVA_HOME="/Applications/Android Studio.app/Contents/jbr/Contents/Home"
export ANDROID_HOME="$HOME/Library/Android/sdk"
export NDK_HOME="$ANDROID_HOME/ndk/25.2.9519653"
export PATH="$PATH:$ANDROID_HOME/emulator"
```

Windows:

```
[System.Environment]::SetEnvironmentVariable("JAVA_HOME", "C:\Program Files\Android\Android Studio\jbr", "User")
[System.Environment]::SetEnvironmentVariable("ANDROID_HOME", "$env:LocalAppData\Android\Sdk", "User")
[System.Environment]::SetEnvironmentVariable("NDK_HOME", "$env:LocalAppData\Android\Sdk\ndk\25.2.9519653", "User")
```

The NDK version in the paths should match the version you installed in the last step

We manually set the PATH variable to include the Android emulator since some distributions of Android Studio include the emulator in the wrong location.

IOS

To develop on IOS, you will need to [install XCode](#). Also make sure to install the iOS targets

```
rustup target add aarch64-apple-ios aarch64-apple-ios-sim
```

If you are using M1, you will have to run `cargo build --target x86_64-apple-ios` instead of `cargo apple build` if you want to run in simulator.

You will also need to install the iOS SDK and the Xcode command line tools.

Running your app

Starting with Dioxus 0.6, dx ships with built-in support for mobile.

Simply create a new Dioxus project:

```
dx new my-app
```

Make sure to launch the relevant mobile simulator. For Android, you can use the Android Studio emulator, or the Android Emulator in the terminal. Make sure to adjust the device name depending on which emulator you installed.

```
emulator -avd Pixel_6_API_34 -netdelay none -netspeed full
```

For iOS, you can use the iOS simulator. You can launch it with:

```
open /Applications/Xcode.app/Contents/Developer/Applications/Simulator.app
xcrun simctl boot "iPhone 15 Pro Max"
```

And then run the app with:

```
cd my-app
dx serve
```

This will start the app in development mode.

Mobile

This guide will cover concepts specific to the Dioxus mobile renderer.

Running Javascript

Dioxus provides some ergonomic wrappers over the browser API, but in some cases you may need to access parts of the browser API Dioxus does not expose.

For these cases, Dioxus desktop exposes the `use_eval` hook that allows you to run raw Javascript in the webview:

src/eval.rs

```
use dioxus::prelude::*;

fn main() {
    launch(app);
}

fn app() -> Element {
    let future = use_resource(move || async move {
        // You can create as many eval instances as you want
        let mut eval = document::eval(
            r#"
            // You can send messages from JavaScript to Rust with the
dioxus.send function
dioxus.send("Hi from JS!");
            // You can receive messages from Rust to JavaScript with the
dioxus.recv function
            let msg = await dioxus.recv();
            console.log(msg);
            "#,
        );

        // You can send messages to JavaScript with the send method
        eval.send("Hi from Rust!").unwrap();

        // You can receive any message from JavaScript with the recv method
        eval.recv::<String>().await.unwrap()
    });

    match future.read_unchecked().as_ref() {
        Some(v) => rsx! {
            p { "{v}" }
        },
        _ => rsx! {
            p { "hello" }
        },
    }
}
```

Custom Assets

You can link to local assets in dioxus mobile instead of using a url:

src/custom_assets.rs

```
use dioxus::prelude::*;

fn main() {
    launch(app);
}

fn app() -> Element {
```

```
rsx! {  
  div {  
    img { src: asset!("/assets/static/scanner.png") }  
  }  
}
```

Integrating with Wry

In cases where you need more low level control over your window, you can use wry APIs exposed through the [Desktop Config](#) and the [use_window hook](#)

Server-Side Rendering

For lower-level control over the rendering process, you can use the `dioxus-ssr` crate directly. This can be useful when integrating with a web framework that `dioxus-fullstack` does not support, or pre-rendering pages.

Setup

For this guide, we're going to show how to use Dioxus SSR with [Axum](#).

Make sure you have Rust and Cargo installed, and then create a new project:

```
cargo new --bin demo  
cd demo
```

Add Dioxus and the ssr renderer as dependencies:

```
cargo add dioxus  
cargo add dioxus-ssr
```

Next, add all the Axum dependencies. This will be different if you're using a different Web Framework

```
cargo add tokio --features full  
cargo add axum
```

Your dependencies should look roughly like this:

```
[dependencies]  
axum = "0.7"  
dioxus = { version = "*" }  
dioxus-ssr = { version = "*" }  
tokio = { version = "1.15.0", features = ["full"] }
```

Now, set up your Axum app to respond on an endpoint.

src/ssr.rs

```
use axum::{response::Html, routing::get, Router};  
use dioxus::prelude::*;  
  
#[tokio::main]  
async fn main() {
```

```

let listener = tokio::net::TcpListener::bind("127.0.0.1:3000")
    .await
    .unwrap();

println!("listening on http://127.0.0.1:3000");

axum::serve(
    listener,
    Router::new()
        .route("/", get(app_endpoint))
        .into_make_service(),
)
    .await
    .unwrap();
}

```

And then add our endpoint. We can either render `rsx!` directly:

src/ssr.rs

```

async fn app_endpoint() -> Html<String> {
    // render the rsx! macro to HTML
    Html(dioxus_ssr::render_element(rsx! { div { "hello world!" } }))
}

```

Or we can render `VirtualDoms`.

src/ssr.rs

```

async fn app_endpoint() -> Html<String> {
    // create a component that renders a div with the text "hello world"
    fn app() -> Element {
        rsx! { div { "hello world" } }
    }
    // create a VirtualDom with the app component
    let mut app = VirtualDom::new(app);
    // rebuild the VirtualDom before rendering
    app.rebuild_in_place();

    // render the VirtualDom to HTML
    Html(dioxus_ssr::render(&app))
}

```

Finally, you can run it using `cargo run` rather than `dx serve`.

Multithreaded Support

The Dioxus `VirtualDom`, sadly, is not currently `Send`. Internally, we use quite a bit of interior mutability which is not thread-safe. When working with web frameworks that require `Send`, it is possible to render a `VirtualDom` immediately to a `String` – but you cannot hold the `VirtualDom` across an `await` point. For retained-state SSR (essentially `LiveView`), you'll need to spawn a `VirtualDom` on its own thread and communicate with it via channels or create a pool of `VirtualDoms`. You might notice that you cannot hold the `VirtualDom` across an `await` point. Because Dioxus is currently not `ThreadSafe`, it *must* remain on the thread it started. We are working on loosening this requirement.

Fullstack development

Dioxus Fullstack contains helpers for:

- Incremental, static, and server side rendering
- Hydrating your application on the Client
- Communicating between a server and a client

This guide will teach you everything you need to know about how to use the utilities in Dioxus fullstack to create amazing fullstack applications.

In addition to this guide, you can find more examples of full-stack apps and information about how to integrate with other frameworks and desktop renderers in the [examples directory](#).

Hydration

In dioxus fullstack, the server renders the initial HTML for improved loading times. This initial version of the page is what most web crawlers and search engines see. After the initial HTML is rendered, the client makes the page interactive and takes over rendering in a process called **hydration**. Most of the time, you shouldn't need to think about hydration, but there are a few things you need to keep in mind to avoid [hydration errors](#). To better understand hydration, let's walk through a simple example:

src/hydration.rs

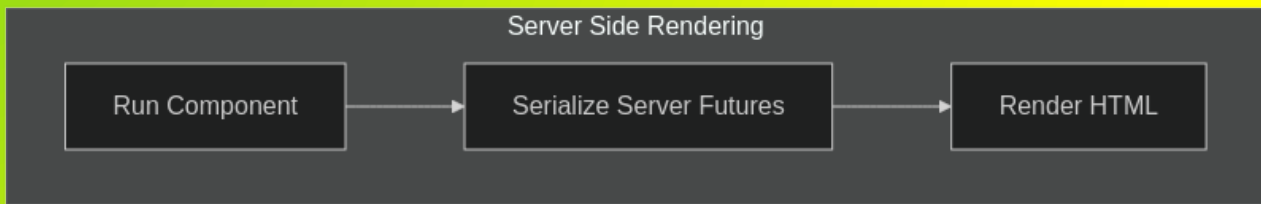
```
fn Weather() -> Element {
    let mut weather = use_server_future(fetch_weather)?;

    rsx! {
        div {
            "{weather:?}"
        }
        button {
            onclick: move |_| weather.restart(),
            "Refetch"
        }
    }
}
```

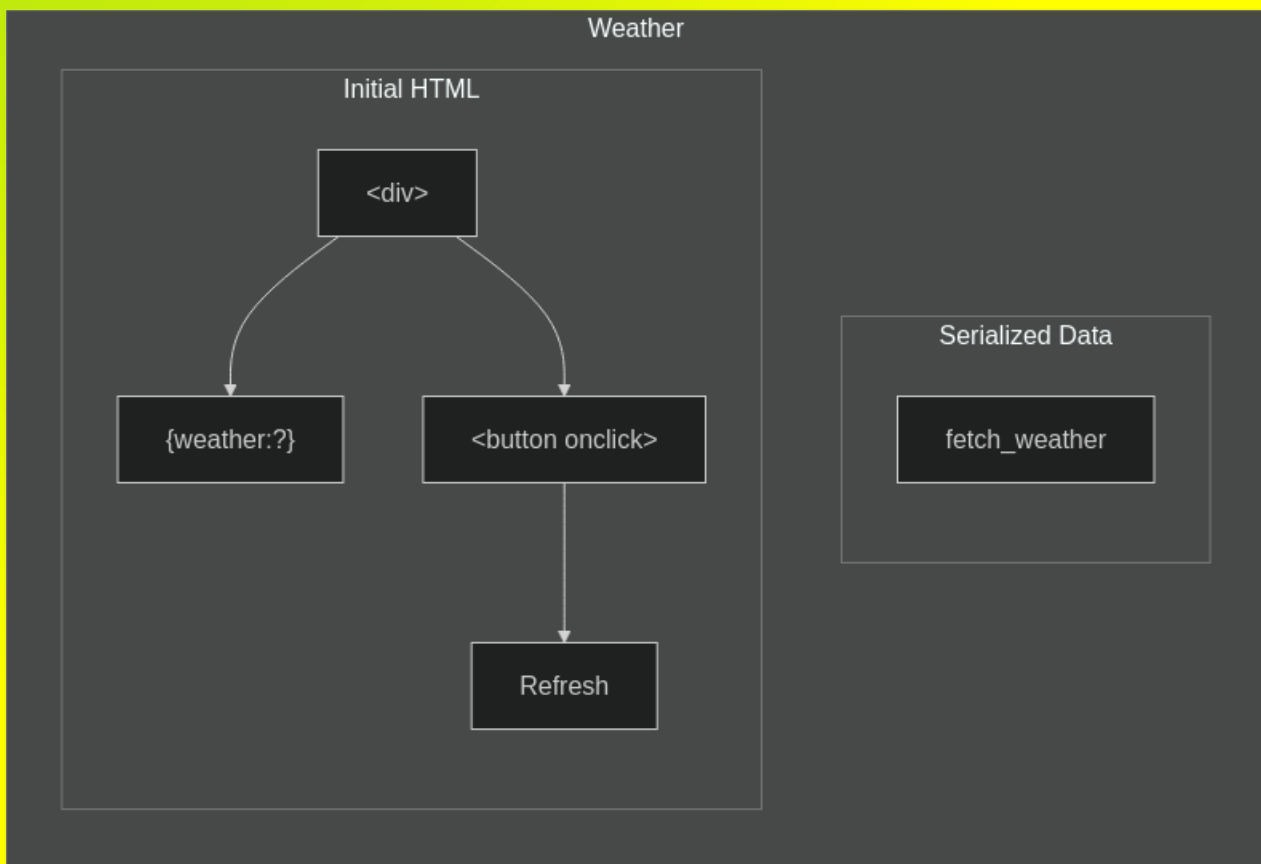
Rendering the initial HTML

When the server receives a request to render the `Weather` component, it renders the page to HTML and serializes some additional data the client needs to hydrate the page. It will follow these steps to render our component:

1. Run the component
2. Wait until all server futures are resolved
3. Serialize any non-deterministic data (like the `weather` future) for the client
4. Render the HTML



Once the server finishes rendering, it will send this structure to the client as HTML:



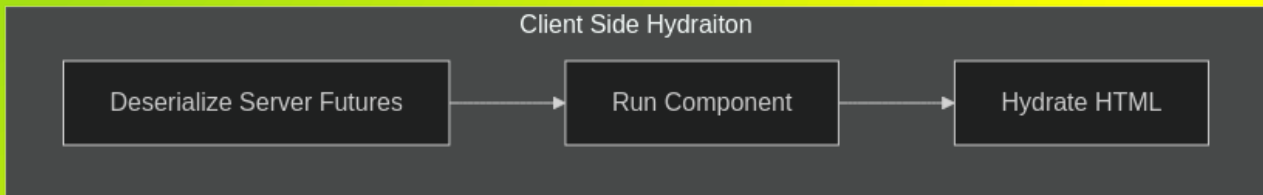
Hydrating on the client

When the client receives the initial HTML, it hydrates the HTML by rerunning each component and linking each node that component renders to the node the server rendered. Rerunning each component lets the client re-construct some non-serializable state like event handlers and kick off any client side logic like `use_effect` and `use_future`.

It will follow these steps:

1. Deserialize any non-deterministic data from the server (like the `weather` future)

2. Run the component with the deserialized data. All server futures are immediately resolved with the deserialized data from the server.
3. Hydrate the HTML sent from the server. This adds all event handlers and links the html nodes to the component so they can be moved or modified later



Hydration errors

For hydration to work, **the component must render exactly the same thing on the client and the server**. If it doesn't, you might see an error like this:

```
Uncaught TypeError: Cannot set properties of undefined (setting 'textContent')
at RawInterpreter.run (yourwasm-hash.js:1:12246)
```

Or this:

```
Error deserializing data:
Semantic(None, "invalid type: floating point `1.2`, expected integer")
This type was serialized on the server at src/main.rs:11:5 with the type name
f64. The client failed to deserialize the type i32 at /path/to/server_future.rs
```

Non-deterministic data with server cached

You must put any non-deterministic data in `use_server_future`, `use_server_cached` or `use_effect` to avoid hydration errors. For example, if you need to render a random number on your page, you can use `use_server_cached` to cache the random number on the server and then use it on the client:

src/hydration.rs

```
// ✗ The random number will be different on the client and the server
let random: u8 = use_hook(|| rand::random());
// ✓ The same random number will be serialized on the server and deserialized
on the client
let random: u8 = use_server_cached(|| rand::random());
```

Async loading with server futures

If you need render some data from a server future, you need to use `use_server_future` to serialize the data instead of waiting for the (non-deterministic) amount of time `use_resource(...).suspend()` takes:

src/hydration.rs

```
// ✗ The server function result may be finished on the server, but pending on the client
let random: u8 = use_resource(|| random_server_function()).suspend()?().unwrap_or_default();
// ✔ Once the server function is resolved on the server, it will be sent to the client
let random: u8 = use_server_future(|| random_server_function())?().unwrap().unwrap_or_default();
```

Client only data with effects

If you need to grab some data that is only available on the client, make sure you get it inside of a `use_effect` hook which runs after the component has been hydrated:

src/hydration.rs

```
// ✗ Using a different value client side before hydration will cause hydration issues
// because the server rendered the html with another value
let mut storage = use_signal(|| {
    #[cfg(feature = "server")]
    return None;
    let window = web_sys::window().unwrap();
    let local_storage = window.local_storage().unwrap().unwrap();
    local_storage.set_item("count", "1").unwrap();
    local_storage.get_item("count").unwrap()
});
// ✔ Changing the value inside of an effect is fine because effects run after hydration
let mut storage = use_signal(|| None);
use_effect(move || {
    let window = web_sys::window().unwrap();
    let local_storage = window.local_storage().unwrap().unwrap();
    local_storage.set_item("count", "1").unwrap();
    storage.set(local_storage.get_item("count").unwrap());
});
```

Avoid side effects in server cached hooks

The dioxus fullstack specific hooks `use_server_cached` and `use_server_future` don't run the same on the server and the client. The server will always run the closure, but the client may not run the closure if the server serialized the result. Because of this, the code you run inside these hooks **cannot have side effects**. If it does, the side effects will not be serialized and it can cause a hydration mismatch error:

src/hydration.rs

```
// ✗ The state of the signal cannot be serialized on the server
let mut storage = use_signal(|| None);
use_server_future(move || async move {
    storage.set(Some(server_future().await));
})?;
// ✔ The value returned from use_server_future will be serialized on the server and hydrated on the client
let storage = use_server_future(|| async move { server_future().await })?;
```

Managing Fullstack Dependencies

Fullstack applications build to at least two different binaries:

- The client application that runs the desktop, mobile, or web application
- The server that renders the initial HTML and runs server functions

Those binaries tend to have different dependencies and those dependencies often are only compatible with a specific target platform. This guide will cover how fullstack manages each binary's dependencies and how to add dependencies that are only compatible with one binary/target.

Client and Server Feature Flags

Dioxus uses feature flags to differentiate between the different binaries a single library can produce. Each target binary should have a feature flag in your `Cargo.toml` file that enables the corresponding feature in dioxus. For example, if you are targeting `web` and `desktop` with a fullstack server, you would add the following to your `Cargo.toml`:

```
[dependencies]
# Don't include any renderer features in your dioxus dependency directly. They
# will be added in feature flags.
# The fullstack feature enables the bindings between the server and client
# without enabling a specific binary target.
dioxus = { version = "0.6", features = ["fullstack"] }

[features]
# The web feature enables the web renderer. Dioxus will automatically enable the
# feature you define that activates `dioxus/web` when building the client WASM
# bundle.
web = ["dioxus/web"]
# The desktop feature enables the desktop renderer. Dioxus will automatically
# enable the feature you define that activates `dioxus/desktop` when building the
# client native bundle.
desktop = ["dioxus/desktop"]
# The server feature enables server functions and server-side rendering. Dioxus
# will automatically enable the feature you define that activates `dioxus/server`
# when building the server binary.
server = ["dioxus/server"]
```

Feature flags like these for the client and server are automatically generated by the CLI when you run `dx new` with fullstack enabled. If you are creating a project from scratch, you will need to add the feature flags manually.

If you are not familiar with features in rust, you can read more about feature flags in the [cargo reference](#).

Adding Server Only Dependencies

Many dependencies like [tokio](#) and [axum](#) are only compatible with the server. If these dependencies are enabled when building a WASM bundle for the browser client, you will get a compilation error. For example, if we want to interact with the filesystem in a server function, we

might want to add `tokio`. `tokio` has utilities for working with async IO like [tokio::fs::File](#). Let's try it as a dependency to our fullstack project:

```
[dependencies]
# ...
# ✗ If tokio is added as a required dependency, it will be included in both the
server
# and the web bundle. The web bundle will fail to build because tokio is not
# compatible with wasm
tokio = { version = "1", features = ["full"] }
```

If we try to compile with `tokio` as a required dependency, we will get a compilation error like this:

```
error[E0432]: unresolved import `crate::sys::IoSourceState`
  --> /Users/user/.cargo/registry/src/index.crates.io-6f17d22bba15001f/mio-
1.0.2/src | source.rs:14:5
14 | use crate::sys::IoSourceState;
    |         ^^^^^^^^^^^^^^^^^^^^^ no `IoSourceState` in `sys`
...

```

Since we added `tokio` as a dependency for all three binaries, cargo tries to compile it for each target. This fails because `tokio` is not compatible with the `wasm32-unknown-unknown` target.

To fix the issue, we can **make the dependency optional and only enable it in the server feature**:

```
[dependencies]
# ...
# ✓ Since the tokio dependency is optional, it is not included in the web and
desktop
# bundles.
tokio = { version = "1", features = ["full"], optional = true }

[features]
# ...
# ✓ Since the tokio dependency is enabled in the server feature, it is included
in
# the server binary.
server = ["dioxus/server", "dep:tokio"]
```

Now when we build with `dx serve`, the project compiles successfully.

Adding Client Only Dependencies

Many dependencies like [wasm-bindgen](#) and [web-sys](#) are only compatible with the client. Unlike server-only dependencies, these dependencies can generally compile on native targets, but they will panic when used outside of the browser.

You can cut down on build times for your server and native binaries by only including web dependencies in the browser client binary.

Instead of adding web only dependencies every binary in your project like this:

```
[dependencies]
# ...
# ✗ If web-sys is added as a required dependency, it will be included in the
server,
# native, and the web bundle which makes build times longer.
web-sys = { version = "0.3.60", features = ["console"] }
```

You can make the dependency optional and only enable it in the `web` feature in your `Cargo.toml`:

```
[dependencies]
# ...
# ✓ Since the web-sys dependency is optional, it is not included in the server
and
# native bundles.
web-sys = { version = "0.3.60", features = ["console"], optional = true }

[features]
# ...
# ✓ Since the web-sys dependency is enabled in the web feature, it is included
in
# the web bundle.
web = ["dioxus/web", "dep:web-sys"]
```

Managing Binary Specific Imports

Once you have set up binary specific dependencies, you need to adjust any of your imports to only import the dependencies when building for the binary that includes those dependencies.

For example, if `tokio` is only enabled in the `server` feature, you will need to import it like this:

`src/server_tokio_import.rs`

```
// Since the tokio dependency is only enabled in the server feature,
// we need to only import it when the server feature is enabled.
#[cfg(feature = "server")]
use tokio::fs::File;
#[cfg(feature = "server")]
use tokio::io::AsyncReadExt;
```

You also need to only compile any usage of the dependency when the feature is enabled:

`src/server_tokio_import.rs`

```
// Since the tokio dependency is only enabled in the server feature,
// we need to only compile any usage of the dependency when the server feature
is enabled.
#[cfg(feature = "server")]
async fn read_file() -> Result<String, std::io::Error> {
    let mut file = File::open("path/to/file").await?;
    let mut contents = String::new();
    file.read_to_string(&mut contents).await?;
    Ok(contents)
}

// The bodies of server functions automatically only compile when the server
feature is enabled.
#[server]
async fn get_file_contents() -> Result<String, ServerFnError> {
    let mut file = File::open("path/to/file").await?;
    let mut contents = String::new();
    file.read_to_string(&mut contents).await?;
    Ok(contents)
}
```

It may be more convenient to group server or client specific code into a module that is only compiled when the feature is enabled:

src/server_tokio_import.rs

```
// Instead of configuring each item that is only used in the server, you can
// group
// them into a module that is only compiled when the server feature is enabled.
#[cfg(feature = "server")]
mod tokio_utilities {
    use std::path::PathBuf;
    use tokio::fs::File;
    use tokio::io::AsyncReadExt;

    pub async fn read_file(path: PathBuf) -> Result<String, std::io::Error> {
        let mut file = File::open(path).await?;
        let mut contents = String::new();
        file.read_to_string(&mut contents).await?;
        Ok(contents)
    }
}

// Then you can define your server functions using shared utilities you defined
// for
// server only code.
#[server]
async fn get_file_contents() -> Result<String, ServerFnError> {
    let file = tokio_utilities::read_file(PathBuf::from("path/to/file")).await?;
    Ok(file)
}

#[server]
async fn get_other_file_contents() -> Result<String, ServerFnError> {
    let file =
tokio_utilities::read_file(PathBuf::from("path/to/other/file")).await?;
    Ok(file)
}
```

The [rust reference](#) has more information about conditional compilation in rust.

Communicating with the server

dioxus-fullstack provides server functions that allow you to call an automatically generated API on the server from the client as if it were a local function.

To make a server function, simply add the `#[server(YourUniqueType)]` attribute to a function. The function must:

- Be an async function
- Have arguments and a return type that both implement `Serialize` and `Deserialize` (with [serde](#)).
- Return a `Result` with an error type of `ServerFnError`

If you are targeting WASM on the server with WASI, you must call `register` on the type you passed into the server macro in your main function before starting your server to tell Dioxus about the server function. For all other targets, the server function will be registered automatically.

Let's continue building on the app we made in the [getting started](#) guide. We will add a server function to our app that allows us to double the count on the server.

First, add `serde` as a dependency:

```
cargo add serde
```

Next, add the server function to your `main.rs`:

`src/server_function.rs`

```
#![allow(non_snake_case)]

use dioxus::prelude::*;

fn main() {
    launch(App)
}

fn App() -> Element {
    let mut count = use_signal(|| 0);

    rsx! {
        h1 { "High-Five counter: {count}" }
        button { onclick: move |_| count += 1, "Up high!" }
        button { onclick: move |_| count -= 1, "Down low!" }
        button {
            onclick: move |_| {
                async move {
                    if let Ok(new_count) = double_server(count()).await {
                        count.set(new_count);
                    }
                },
                "Double"
            }
        }
    }
}

#[server]
async fn double_server(number: i32) -> Result<i32, ServerFnError> {
    // Perform some expensive computation or access a database on the server
    tokio::time::sleep(std::time::Duration::from_secs(1)).await;
    let result = number * 2;
    println!("server calculated {result}");
    Ok(result)
}
```

Now, build your client-side bundle with `dx build --features web` and run your server with `cargo run --features ssr`. You should see a new button that multiplies the count by 2.

Cached data fetching

One common use case for server functions is fetching data from the server:

`src/server_data_fetch.rs`

```
#![allow(non_snake_case, unused)]

use dioxus::prelude::*;

fn main() {
    launch(app)
}
```

```
fn app() -> Element {
    let mut count = use_resource(get_server_data);

    rsx! {"server data is {count.value():?}"}
}

#[server]
async fn get_server_data() -> Result<String, ServerFnError> {
    // Access a database
    tokio::time::sleep(std::time::Duration::from_millis(100)).await;
    Ok("Hello from the server!".to_string())
}
```

If you navigate to the site above, you will first see `server data is None`, then after the WASM has loaded and the request to the server has finished, you will see `server data is Some(Ok("Hello from the server!"))`.

This approach works, but it can be slow. Instead of waiting for the client to load and send a request to the server, what if we could get all of the data we needed for the page on the server and send it down to the client with the initial HTML page?

This is exactly what the `use_server_future` hook allows us to do! `use_server_future` is similar to the `use_resource` hook, but it allows you to wait for a future on the server and send the result of the future down to the client.

Let's change our data fetching to use `use_server_future`:

src/server_data_prefetch.rs

```
#![allow(non_snake_case, unused)]

use dioxus::prelude::*;

fn main() {
    launch(app);
}

fn app() -> Element {
    let mut count = use_server_future(get_server_data)?;

    rsx! {"server data is {count.value():?}"}
}

#[server]
async fn get_server_data() -> Result<String, ServerFnError> {
    // Access a database
    tokio::time::sleep(std::time::Duration::from_millis(100)).await;
    Ok("Hello from the server!".to_string())
}
```

Notice the `?` after `use_server_future`. This is what tells Dioxus fullstack to wait for the future to resolve before continuing rendering. If you want to not wait for a specific future, you can just remove the `?` and deal with the `Option` manually.

Now when you load the page, you should see `server data is Ok("Hello from the server!")`. No need to wait for the WASM to load or wait for the request to finish!

Running the client with dioxus-desktop

The project presented so far makes a web browser interact with the server, but it is also possible to make a desktop program interact with the server in a similar fashion. (The full example code is available in the [Dioxus repo](#))

First, we need to make two binary targets, one for the desktop program (the `client.rs` file), one for the server (the `server.rs` file). The client app and the server functions are written in a shared `lib.rs` file.

The desktop and server targets have slightly different build configuration to enable additional dependencies or features. The Cargo.toml in the full example has more information, but the main points are:

- the `client.rs` has to be run with the `desktop` feature, so that the optional `dioxus-desktop` dependency is included
- the `server.rs` has to be run with the `ssr` features; this will generate the server part of the server functions and will run our backend server.

Once you create your project, you can run the server executable with:

```
cargo run --bin server --features ssr
```

and the client desktop executable with:

```
cargo run --bin client --features desktop
```

Client code

The client file is pretty straightforward. You only need to set the server url in the client code, so it knows where to send the network requests. Then, `dioxus_desktop` launches the app.

For development, the example project runs the server on `localhost:8080`. **Before you release remember to update the url to your production url.**

Server code

In the server code, first you have to set the network address and port where the server will listen to.

`src/server_function_desktop_client.rs`

```
let listener = tokio::net::TcpListener::bind("127.0.0.1:3000")
    .await
    .unwrap();
println!("listening on http://127.0.0.1:3000");
```

Then, you have to register the types declared in the server function macros into the server. For example, consider this server function:

`src/server_function_desktop_client.rs`

```
#[server(GetServerData)]
async fn get_server_data() -> Result<String, ServerFnError> {
    Ok("Hello from the server!".to_string())
}
```

The `GetServerData` type has to be registered in the server, which will add the corresponding route to the server.

`src/server_function_desktop_client.rs`

Finally, the server is started and it begins responding to requests.

Extractors

Server functions are an ergonomic way to call a function on the server. Server function work by registering an endpoint on the server and using requests on the client. Most of the time, you shouldn't need to worry about how server functions operate, but there are some times when you need to get some value from the request other than the data passed in the server function.

For example, requests contain information about the user's browser (called the [user agent](#)). We can use an extractor to retrieve that information.

You can use the `extract` method within a server function to extract something from the request. You can extract any type that implements `FromServerContext` (or when axum is enabled, you can use axum extractors directly):

`src/server_function_extract.rs`

```
// #[server]
// pub async fn log_headers() -> Result<(), ServerFnError> {
//     let headers: http::HeaderMap = extract().await?;
//     log::info!("{:?}", headers[http::header::USER_AGENT]);
//     Ok(())
// }
```

Middleware

Extractors allow you to wrap your server function in some code that changes either the request or the response. Dioxus fullstack integrates with [Tower](#) to allow you to wrap your server functions in middleware.

You can use the `#[middleware]` attribute to add a layer of middleware to your server function. Let's add a timeout middleware to a server function. This middleware will stop running the server function if it reaches a certain timeout:

`src/server_function_middleware.rs`

```
// #[server]
// // Add a timeout middleware to the server function that will return an error
// if the function takes longer than 1 second to execute
//
// #[middleware(tower_http::timeout::TimeoutLayer::new(std::time::Duration::from_secs(1)))]
// pub async fn timeout() -> Result<(), ServerFnError> {
//     tokio::time::sleep(std::time::Duration::from_secs(2)).await;
//     Ok(())
// }
```

Authentication

You can use [extractors](#) to integrate auth with your Fullstack application.

You can create a custom extractors that extracts the auth session from the request. From that auth session, you can check if the user has the required privileges before returning the private data.

A [full auth example](#) with the complete implementation is available in the fullstack examples.

Routing

You can easily integrate your fullstack application with a client side router using Dioxus Router. This allows you to create different scenes in your app and navigate between them. You can read more about the router in the [router reference](#)

src/server_router.rs

```
#![allow(non_snake_case)]

use axum::Router;
use dioxus::prelude::*;

use dioxus_router::prelude::*;
use serde::{Deserialize, Serialize};

fn main() {
    launch(|| rsx! { Router::<Route> {} });
}

#[derive(Clone, Routable, Debug, PartialEq, Serialize, Deserialize)]
enum Route {
    #[route("/")]
    Home {},
    #[route("/blog/:id")]
    Blog { id: i32 },
}

#[component]
fn Blog(id: i32) -> Element {
    rsx! {
        Link { to: Route::Home {}, "Go to counter" }
        table {
            tbody {
                for _ in 0..id {
                    tr {
                        for _ in 0..id {
                            td { "hello world!" }
                        }
                    }
                }
            }
        }
    }
}

#[component]
fn Home() -> Element {
```

```

let mut count = use_signal(|| 0);
let mut text = use_signal(|| "...".to_string());

rsx! {
  Link { to: Route::Blog { id: count() }, "Go to blog" }
  div {
    h1 { "High-Five counter: {count}" }
    button { onclick: move |_| count += 1, "Up high!" }
    button { onclick: move |_| count -= 1, "Down low!" }
    button {
      onclick: move |_| {
        async move {
          if let Ok(data) = get_server_data().await {
            println!("Client received: {}", data);
            text.set(data.clone());
            post_server_data(data).await.unwrap();
          }
        }
      },
      "Run server function!"
    }
    "Server said: {text}"
  }
}

#[server(PostServerData)]
async fn post_server_data(data: String) -> Result<(), ServerFnError> {
  println!("Server received: {}", data);

  Ok(())
}

#[server(GetServerData)]
async fn get_server_data() -> Result<String, ServerFnError> {
  Ok("Hello from the server!".to_string())
}

```

Static Site Generation

Static site generation (SSG) lets you pre-generate all static pages of your application at build time. Once you have the static html pages, you can deploy them to any static hosting provider like GitHub Pages.

Setting up the ServeConfig

SSG builds on top of the incremental rendering feature of Dioxus fullstack. We need to set up the `ServeConfig` to enable incremental rendering. The incremental config needs to render to the `public` directory where Dioxus places all other public files like the wasm binary and static assets. The `public` directory in the web folder will always be placed alongside the server binary.

src/static_site_generation.rs

```

fn main() {
  dioxus::LaunchBuilder::new()
    // Set the server config only if we are building the server target
    .with_cfg(server_only! {
      ServeConfig::builder()
    })
}

```

```

        // Enable incremental rendering
        .incremental(
            IncrementalRendererConfig::new()
            // Store static files in the public directory where
other static assets like wasm are stored
            .static_dir(
                std::env::current_exe()
                    .unwrap()
                    .parent()
                    .unwrap()
                    .join("public")
            )
            // Don't clear the public folder on every build. The
public folder has other files including the wasm
            // binary and static assets required for the app to run
            .clear_cache(false)
        )
        .enable_out_of_order_streaming()
    })
    .launch(app);
}

```

Configuring static routes

Once you have incremental rendering enabled, you need to tell the CLI about the static routes in your app. The CLI looks for a server function at the endpoint "static_routes" that returns a list of all static urls. It will call this server function at build time and pre-render all of the routes in the list.

src/static_site_generation.rs

```

#[derive(Routable, Clone, PartialEq)]
pub enum Route {
    // Any routes with no dynamic segments in your router will be included in
the static routes list
    #[route("/")]
    Index {},
    #[route("/other")]
    Other {},
}

// The server function at the endpoint "static_routes" will be called by the CLI
to generate the list of static
// routes. You must explicitly set the endpoint to `static_routes` in the
server function attribute instead of
// the default randomly generated endpoint.
#[server(endpoint = "static_routes")]
async fn static_routes() -> Result<Vec<String>, ServerFnError> {
    // The `Routable` trait has a `static_routes` method that returns all static
routes in the enum
    Ok(Route::static_routes().iter().map(ToString::to_string).collect())
}

```

Publishing static sites

Finally, you can bundle your site with `dx bundle --platform web --ssg`. Once the CLI finishes bundling, you should see a `public` folder in the `dx` folder of your project:

The folder contains all of the static assets that you need to serve your site. You can copy the public folder into any static hosting provider like GitHub Pages.

Publishing

After you have build your application, you will need to publish it somewhere. This reference will outline different methods of publishing your desktop or web application.

Web: Publishing with GitHub Pages

Edit your `Dioxus.toml` to point your `out_dir` to the `docs` folder and the `base_path` to the name of your repo:

```
[application]
# ...
out_dir = "docs"

[web.app]
base_path = "your_repo"
```

Then build your app and publish it to Github:

- Make sure GitHub Pages is set up for your repo to publish any static files in the docs directory
- Build your app with:

```
dx build --release
```

- Make a copy of your `docs/index.html` file and rename the copy to `docs/404.html` so that your app will work with client-side routing
- Add and commit with git
- Push to GitHub

Desktop: Creating an installer

Dioxus desktop app uses your operating system's WebView library, so it's portable to be distributed for other platforms.

In this section, we'll cover how to bundle your app for macOS, Windows, and Linux.

Preparing your application for bundling

Depending on your platform, you may need to add some additional code to your `main.rs` file to make sure your app is ready for bundling. On Windows, you'll need to add the `#!`

`[windows_subsystem = "windows"]` attribute to your `main.rs` file to hide the terminal window that pops up when you run your app. **If you're developing on Windows, only use this when bundling.** It will disable the terminal, so you will not get logs of any kind. You can gate it behind a feature, like so:

```
# Cargo.toml
```

```
[features]
bundle = []
```

And then your `main.rs`:

```
#![cfg_attr(feature = "bundle", windows_subsystem = "windows")]
```

[Adding assets to your application](#)

If you want to bundle assets with your application, you can either use them with the `manganis` crate (covered more in the [assets](#) page), or you can include them in your `Dioxus.toml` file:

```
[bundle]
# The list of files to include in the bundle. These can contain globs.
resources = ["main.css", "header.svg", "**/*.png"]
```

[Install dioxus CLI](#)

The first thing we'll do is install the [dioxus-cli](#). This extension to cargo will make it very easy to package our app for the various platforms.

To install, simply run

```
cargo install dioxus-cli
```

[Building](#)

To bundle your application you can simply run `dx bundle --release` (also add `--features bundle` if you're using that, see the [this](#) for more) to produce a final app with all the optimizations and assets builtin.

Once you've ran the command, your app should be accessible in `dist/bundle/`.

For example, a macOS app would look like this:

Nice! And it's only 4.8 Mb – extremely lean!! Because Dioxus leverages your platform's native WebView, Dioxus apps are extremely memory efficient and won't waste your battery.

Note: not all CSS works the same on all platforms. Make sure to view your app's CSS on each platform – or web browser (Firefox, Chrome, Safari) before publishing.

[Antipatterns](#)

This example shows what not to do and provides a reason why a given pattern is considered an "AntiPattern". Most anti-patterns are considered wrong for performance or code re-usability reasons.

Unnecessarily Nested Fragments

Fragments don't mount a physical element to the DOM immediately, so Dioxus must recurse into its children to find a physical DOM node. This process is called "normalization". This means that deeply nested fragments make Dioxus perform unnecessary work. Prefer one or two levels of fragments / nested components until presenting a true DOM element.

Only Component and Fragment nodes are susceptible to this issue. Dioxus mitigates this with components by providing an API for registering shared state without the Context Provider pattern.

src/anti_patterns.rs

```
// ✗ Don't unnecessarily nest fragments
let _ = rsx! {
    Fragment {
        Fragment {
            Fragment {
                Fragment {
                    Fragment {
                        Fragment { div { "Finally have a real node!" } }
                    }
                }
            }
        }
    }
};

// ✔ Render shallow structures
rsx! { div { "Finally have a real node!" } }
```

Incorrect Iterator Keys

As described in the [dynamic rendering chapter](#), list items must have unique keys that are associated with the same items across renders. This helps Dioxus associate state with the contained components and ensures good diffing performance. Do not omit keys, unless you know that the list will never change.

src/anti_patterns.rs

```
let data: &HashMap<_, _> = &props.data;

// ✗ No keys
rsx! {
    ul {
        for value in data.values() {
            li { "List item: {value}" }
        }
    }
};

// ✗ Using index as keys
rsx! {
    ul {
        for (index, value) in data.values().enumerate() {
            li { key: "{index}", "List item: {value}" }
        }
    }
};

// ✔ Using unique IDs as keys:
rsx! {
```

```

    ul {
      for (key, value) in props.data.iter() {
        li { key: "{key}", "List item: {value}" }
      }
    }
  }
}

```

Avoid Interior Mutability in Props

While it is technically acceptable to have a `Mutex` or a `RwLock` in the props, they will be difficult to use.

Suppose you have a struct `User` containing the field `username: String`. If you pass a `Mutex<User>` prop to a `UserComponent` component, that component may wish to write to the `username` field. However, when it does, the parent component will not be aware of the change, and the component will not re-render which causes the UI to be out of sync with the state. Instead, consider passing down a reactive value like a `Signal` or immutable data.

src/anti_patterns.rs

```

// ✗ Mutex/RwLock/RefCell in props
#[derive(Props, Clone)]
struct AntipatternInteriorMutability {
    map: Rc<RefCell<HashMap<u32, String>>>,
}

impl PartialEq for AntipatternInteriorMutability {
    fn eq(&self, other: &Self) -> bool {
        std::rc::Rc::ptr_eq(&self.map, &other.map)
    }
}

fn AntipatternInteriorMutability(map: Rc<RefCell<HashMap<u32, String>>>) ->
Element {
    rsx! {
        button {
            onclick: {
                let map = map.clone();
                move |_| {
                    // Writing to map will not rerun any components
                    map.borrow_mut().insert(0, "Hello".to_string());
                }
            },
            "Mutate map"
        }
    }
    // Since writing to map will not rerun any components, this will get out
of date
    "{map.borrow().get(&0).unwrap()}"
}

// ✓ Use a signal to pass mutable state
#[component]
fn AntipatternInteriorMutabilitySignal(map: Signal<HashMap<u32, String>>) ->
Element {
    rsx! {
        button {
            onclick: move |_| {
                // Writing to map will rerun any components that read the map
                map.write().insert(0, "Hello".to_string());
            }
        }
    }
}

```

```

    },
    "Mutate map"
  }
  // Since writing to map will rerun subscribers, this will get updated
  "{map.read().get(&0).unwrap()}"
}
}

```

Avoid Updating State During Render

Every time you update the state, Dioxus needs to re-render the component – this is inefficient! Consider refactoring your code to avoid this.

Also, if you unconditionally update the state during render, it will be re-rendered in an infinite loop.

src/anti_patterns.rs

```

// ✗ Updating state in render
let first_signal = use_signal(|| 0);
let mut second_signal = use_signal(|| 0);

// Updating the state during a render can easily lead to infinite loops
if first_signal() + 1 != second_signal() {
    second_signal.set(first_signal() + 1);
}

// ✓ Update state in an effect
let first_signal = use_signal(|| 0);
let mut second_signal = use_signal(|| 0);

// The closure you pass to use_effect will be rerun whenever any of the
dependencies change without re-rendering the component
use_effect(move || {
    if first_signal() + 1 != second_signal() {
        second_signal.set(first_signal() + 1);
    }
});

// ✓ Deriving state with use_memo
let first_signal = use_signal(|| 0);
// Memos are specifically designed for derived state. If your state fits this
pattern, use it.
let second_signal = use_memo(move || first_signal() + 1);

```

Avoid Large Groups of State

It can be tempting to have a single large state struct that contains all of your application's state. However, this can lead to issues:

- It can be easy to accidentally mutate the state in a way that causes an infinite loop
- It can be difficult to reason about when and how the state is updated
- It can lead to performance issues because many components will need to re-render when the state changes

Instead, consider breaking your state into smaller, more manageable pieces. This will make it easier to reason about the state, avoid update loops, and improve performance.

src/anti_patterns.rs

```

fn app() -> Element {
  // ✗ Large state struct
  #[derive(Props, Clone, PartialEq)]
  struct LargeState {
    users: Vec<User>,
    logged_in: bool,
    warnings: Vec<String>,
  }

  #[derive(Props, Clone, PartialEq)]
  struct User {
    name: String,
    email: String,
  }

  let mut all_my_state = use_signal(|| LargeState {
    users: vec![User {
      name: "Alice".to_string(),
      email: "alice@example.com".to_string(),
    }],
    logged_in: true,
    warnings: vec![],
  });

  use_effect(move || {
    // It is very easy to accidentally read and write to the state object if
    it contains all your state
    let read = all_my_state.read();
    let logged_in = read.logged_in;
    if !logged_in {
      all_my_state
        .write_unchecked()
        .warnings
        .push("You are not logged in".to_string());
    }
  });

  // ✓ Use multiple signals to manage state
  let users = use_signal(|| {
    vec![User {
      name: "Alice".to_string(),
      email: "alice@example.com".to_string(),
    }]
  });
  let logged_in = use_signal(|| true);
  let mut warnings = use_signal(|| vec![]);

  use_effect(move || {
    // Now you can read and write to separate signals which will not cause
    issues
    if !logged_in() {
      warnings.write().push("You are not logged in".to_string());
    }
  });

  // ✓ Use memos to create derived state when larger states are unavoidable
  // Notice we didn't split everything into separate signals. Users still make
  sense as a vec of data
  let users = use_signal(|| {
    vec![User {
      name: "Alice".to_string(),
      email: "alice@example.com".to_string(),
    }]
  });
}

```

```

let logged_in = use_signal(|| true);
let warnings: Signal<Vec<String>> = use_signal(|| vec![]);

// In child components, you can use the memo to create derived that will
only update when a specific part of the state changes
// This will help you avoid unnecessary re-renders and infinite loops
#[component]
fn FirstUser(users: Signal<Vec<User>>) -> Element {
    let first_user = use_memo(move ||
users.read().first().unwrap().clone());

    rsx! {
        div {
            "First user: {first_user().name}"
        }
    }
}

rsx! {
    FirstUser {
        users
    }
}
}

```

Running Non-Deterministic Code in the Body of a Component

If you have a component that contains non-deterministic code, that code should generally not be run in the body of the component. If it is put in the body of the component, it will be executed every time the component is re-rendered which can lead to performance issues.

Instead, consider moving the non-deterministic code into a hook that only runs when the component is first created or an effect that reruns when dependencies change.

src/anti_patterns.rs

```

// ✗ Non-deterministic code in the body of a component
#[component]
fn NonDeterministic(name: String) -> Element {
    let my_random_id = rand::random::<u64>();

    rsx! {
        div {
            // Id will change every single time the component is re-rendered
            id: "{my_random_id}",
            "Hello {name}"
        }
    }
}

// ✓ Use a hook to run non-deterministic code
fn NonDeterministicHook(name: String) -> Element {
    // If you store the result of the non-deterministic code in a hook, it will
    stay the same between renders
    let my_random_id = use_hook(|| rand::random::<u64>());

    rsx! {
        div {
            id: "{my_random_id}",
            "Hello {name}"
        }
    }
}

```

```
}  
}
```

Overly Permissive PartialEq for Props

You may have noticed that `Props` requires a `PartialEq` implementation. That `PartialEq` is very important for `Dioxus` to work correctly. It is used to determine if a component should re-render or not when the parent component re-renders.

If you cannot derive `PartialEq` for your `Props`, you will need to implement it yourself. If you do implement `PartialEq`, make sure to return `false` any time the props change in a way that would cause the UI in the child component to change.

In general, returning `false` from `PartialEq` if you aren't sure if the props have changed or not is better than returning `true`. This will help you avoid out of date UI in your child components.

src/anti_patterns.rs

```
// ✗ Permissive PartialEq for Props  
#[derive(Props, Clone)]  
struct PermissivePartialEqProps {  
    name: String,  
}  
  
// This will cause the component to **never** re-render when the parent  
// component re-renders  
impl PartialEq for PermissivePartialEqProps {  
    fn eq(&self, _: &Self) -> bool {  
        true  
    }  
}  
  
fn PermissivePartialEq(name: PermissivePartialEqProps) -> Element {  
    rsx! {  
        div {  
            "Hello {name.name}"  
        }  
    }  
}  
  
#[component]  
fn PermissivePartialEqParent() -> Element {  
    let name = use_signal(|| "Alice".to_string());  
  
    rsx! {  
        PermissivePartialEq {  
            // The PermissivePartialEq component will not get the updated value  
            // of name because the PartialEq implementation says that the props are the same  
            name: name()  
        }  
    }  
}  
  
// ✓ Derive PartialEq for Props  
#[derive(Props, Clone, PartialEq)]  
struct DerivePartialEqProps {  
    name: String,  
}  
  
fn DerivePartialEq(name: DerivePartialEqProps) -> Element {
```

```

    rsx! {
        div {
            "Hello {name.name}"
        }
    }
}

#[component]
fn DerivePartialEqParent() -> Element {
    let name = use_signal(|| "Alice".to_string());

    rsx! {
        DerivePartialEq {
            name: name()
        }
    }
}

// ✓ Return false from PartialEq if you are unsure if the props have changed
#[derive(Debug)]
struct NonPartialEq;

#[derive(Props, Clone)]
struct RcPartialEqProps {
    name: Rc<NonPartialEq>,
}

impl PartialEq for RcPartialEqProps {
    fn eq(&self, other: &Self) -> bool {
        // This will almost always return false because the Rc will likely point
        // to a different value
        // Implementing PartialEq for NonPartialEq would be better, but if it is
        // controlled by another library, it may not be possible
        // **Always** return false if you are unsure if the props have changed
        std::rc::Rc::ptr_eq(&self.name, &other.name)
    }
}

fn RcPartialEq(name: RcPartialEqProps) -> Element {
    rsx! {
        div {
            "Hello {name.name:?}"
        }
    }
}

fn RcPartialEqParent() -> Element {
    let name = use_signal(|| Rc::new(NonPartialEq));

    rsx! {
        RcPartialEq {
            // Generally, RcPartialEq will rerun even if the value of name
            // hasn't actually changed because the Rc will point to a different value
            name: name()
        }
    }
}

```

This section of the guide provides getting started guides for common tools used with Dioxus.

- [Logging](#)
- [Internationalization](#)

Logging

Dioxus has a wide range of supported platforms, each with their own logging requirements. We'll discuss the different options available for your projects.

Dioxus Logger

Dioxus provides a first-party logger as part of `launch`. This sets up a tracing subscriber that cleanly integrates with the Dioxus CLI and platforms like Web and Mobile. In development mode, the `Debug` tracing level is set, and in release only the `Info` level is set.

```
use dioxus::prelude::*;

fn main() {
    dioxus::launch(|| {
        // Will only log in "dev" mode
        tracing::debug!("Rendering app!");

        // Will log in dev and release
        tracing::info!("Rendering app!");

        rsx! {}
    })
}
```

To override the default or initialize the logger before `launch`, you can use the `init` function yourself:

To use Dioxus Logger, call the `init()` function:

```
use tracing::Level;

fn main() {
    // Init logger
    dioxus_logger::init(Level::INFO).expect("failed to init logger");

    // Dioxus launch code
    dioxus::launch(|| rsx! {})
}
```

The Tracing Crate

The [Tracing](#) crate is the logging interface that the `dioxus-logger` uses. It is not required to use the Tracing crate, but you will not receive logs from the Dioxus library.

The Tracing crate provides a variety of simple `println`-like macros with varying levels of severity. The available macros are as follows with the highest severity on the bottom:

```
fn main() {
```

```
tracing::trace!("trace");
tracing::debug!("debug");
tracing::info!("info");
tracing::warn!("warn");
tracing::error!("error");
}
```

All the loggers provided on this page are, besides configuration and initialization, interfaced using these macros. Often you will also utilize the Tracing crate's `Level` enum. This enum usually represents the maximum log severity you want your application to emit and can be loaded from a variety of sources such as configuration file, environment variable, and more.

For more information, visit the Tracing crate's [docs](#).

Platform Intricacies

On web, Dioxus Logger will use [tracing-wasm](#). On Desktop and server-based targets, Dioxus Logger will use [tracing-subscriber](#)'s `FmtSubscriber`.

Viewing Logs

Android logs are sent to logcat. To use logcat through the Android debugger, run:

```
adb -d logcat
```

Your Android device will need developer options/usb debugging enabled.

For more information, visit `android_logger`'s [docs](#).

iOS logs are sent to oslog.

For more information, visit [oslog](#).

Final Notes

Dioxus Logger is the preferred logger to use with Dioxus if it suites your needs. There are more features to come. If there are any feature suggestions or issues with Dioxus Logger, feel free to reach out on the [Dioxus Discord Server](#)!

For more information, visit Dioxus Logger's [docs](#).

Internationalization

If your application supports multiple languages, the [dioxus-i18n](#) crate contains helpers to make working with translations in your application easier.

You can find an example [here](#).

State Cookbook

- [External State](#)
- [Custom Hook](#)

Working with External State

This guide will help you integrate your Dioxus application with some external state like a different thread or a websocket connection.

Working with non-reactive State

[Coroutines](#) are great tool for dealing with non-reactive (state you don't render directly) state within your application.

You can store your state inside the coroutine async block and communicate with the coroutine with messages from any child components.

src/use_coroutine.rs

```
// import futures::StreamExt to use the next() method
use futures::StreamExt;
let mut response_state = use_signal(|| None);
let tx = use_coroutine(move |mut rx| async move {
    // Define your state before the loop
    let mut state = reqwest::Client::new();
    let mut cache: HashMap<String, String> = HashMap::new();
    loop {
        // Loop and wait for the next message
        if let Some(request) = rx.next().await {
            // Resolve the message
            let response = if let Some(response) = cache.get(&request) {
                response.clone()
            } else {
                let response = state
                    .get(&request)
                    .send()
                    .await
                    .unwrap()
                    .text()
                    .await
                    .unwrap();
                cache.insert(request, response.clone());
                response
            };
            response_state.set(Some(response));
        } else {
            break;
        }
    }
});
// Send a message to the coroutine
tx.send("https://example.com".to_string());
// Get the current state of the coroutine
let response = response_state.read();
```

Making Reactive State External

If you have some reactive state (state that is rendered), that you want to modify from another thread, you can use a signal that is sync. Signals take an optional second generic value with information about syncness. Sync signals have a slightly higher overhead than thread local signals, but they can be used in a multithreaded environment.

src/sync_signal.rs

```
use dioxus::prelude::*;

fn main() {
    launch(app);
}

fn app() -> Element {
    let mut signal = use_signal_sync(|| 0);

    use_hook(|| {
        std::thread::spawn(move || loop {
            std::thread::sleep(std::time::Duration::from_secs(1));
            // You can easily update the signal from a different thread
            signal += 1;
        });
    });

    rsx! {
        button { onclick: move |_| signal += 1, "Increase" }
        "{signal}"
    }
}
```

Custom Hooks

Hooks are a great way to encapsulate business logic. If none of the existing hooks work for your problem, you can write your own.

When writing your hook, you can make a function that starts with `use_` and takes any arguments you need. You can then use the `use_hook` method to create a hook that will be called the first time the component is rendered.

Composing Hooks

To avoid repetition, you can encapsulate business logic based on existing hooks to create a new hook.

For example, if many components need to access an `AppSettings` struct, you can create a "shortcut" hook:

src/hooks_composed.rs

```
fn use_settings() -> Signal<AppSettings> {
    consume_context()
}
```

Or if you want to wrap a hook that persists reloads with the storage API, you can build on top of the `use_signal` hook to work with mutable state:

src/hooks_composed.rs

```
use gloo_storage::{LocalStorage, Storage};
use serde::{de::DeserializeOwned, Serialize};
```

```

/// A persistent storage hook that can be used to store data across application
reloads.
#[allow(clippy::needless_return)]
pub fn use_persistent<T: Serialize + DeserializeOwned + Default + 'static>(
    // A unique key for the storage entry
    key: impl ToString,
    // A function that returns the initial value if the storage entry is empty
    init: impl FnOnce() -> T,
) -> UsePersistent<T> {
    // Use the use_signal hook to create a mutable state for the storage entry
    let state = use_signal(move || {
        // This closure will run when the hook is created
        let key = key.to_string();
        let value = LocalStorage::get(key.as_str()).ok().unwrap_or_else(init);
        StorageEntry { key, value }
    });

    // Wrap the state in a new struct with a custom API
    UsePersistent { inner: state }
}

struct StorageEntry<T> {
    key: String,
    value: T,
}

/// Storage that persists across application reloads
pub struct UsePersistent<T: 'static> {
    inner: Signal<StorageEntry<T>>,
}

impl<T> Clone for UsePersistent<T> {
    fn clone(&self) -> Self {
        *self
    }
}

impl<T> Copy for UsePersistent<T> {}

impl<T: Serialize + DeserializeOwned + Clone + 'static> UsePersistent<T> {
    /// Returns a reference to the value
    pub fn get(&self) -> T {
        self.inner.read().value.clone()
    }

    /// Sets the value
    pub fn set(&mut self, value: T) {
        let mut inner = self.inner.write();
        // Write the new value to local storage
        LocalStorage::set(inner.key.as_str(), &value);
        inner.value = value;
    }
}

```

Custom Hook Logic

You can use [use_hook](#) to build your own hooks. In fact, this is what all the standard hooks are built on!

`use_hook` accepts a single closure for initializing the hook. It will be only run the first time the component is rendered. The return value of that closure will be used as the value of the hook –

Dioxus will take it, and store it for as long as the component is alive. On every render (not just the first one!), you will get a reference to this value.

Note: You can use the `use_on_destroy` hook to clean up any resources the hook uses when the component is destroyed.

Inside the initialization closure, you will typically make calls to other `CX` methods. For example:

- The `use_signal` hook tracks state in the hook value, and uses [schedule_update](#) to make Dioxus re-render the component whenever it changes.

Here is a simplified implementation of the `use_signal` hook:

src/hooks_custom_logic.rs

```
use std::cell::RefCell;
use std::rc::Rc;
use std::sync::Arc;

struct Signal<T> {
    value: Rc<RefCell<T>>,
    update: Arc<dyn Fn()>,
}

impl<T> Clone for Signal<T> {
    fn clone(&self) -> Self {
        Self {
            value: self.value.clone(),
            update: self.update.clone(),
        }
    }
}

fn my_use_signal<T: 'static>(init: impl FnOnce() -> T) -> Signal<T> {
    use_hook(|| {
        // The update function will trigger a re-render in the component cx is
        // attached to
        let update = schedule_update();
        // Create the initial state
        let value = Rc::new(RefCell::new(init()));

        Signal { value, update }
    })
}

impl<T: Clone> Signal<T> {
    fn get(&self) -> T {
        self.value.borrow().clone()
    }

    fn set(&self, value: T) {
        // Update the state
        *self.value.borrow_mut() = value;
        // Trigger a re-render on the component the state is from
        (self.update)();
    }
}
```

- The `use_context` hook calls [consume_context](#) (which would be expensive to call on every render) to get some context from the component

Here is an implementation of the `use_context` and `use_context_provider` hooks:

`src/hooks_custom_logic.rs`

```
pub fn use_context<T: 'static + Clone>() -> T {
    use_hook(|| consume_context())
}

pub fn use_context_provider<T: 'static + Clone>(f: impl FnOnce() -> T) -> T {
    use_hook(|| {
        let val = f();
        // Provide the context state to the component
        provide_context(val.clone());
        val
    })
}
```

Bundling config

The `[bundle]` section of our `Dioxus.toml` can take a variety of options.

Here are the options, in the form of Rust structs.

```
#[derive(Debug, Clone, Serialize, Deserialize, Default)]
pub(crate) struct BundleConfig {
    /// eg. com.dioxuslabs
    pub(crate) identifier: Option<String>,
    /// eg. DioxusLabs
    pub(crate) publisher: Option<String>,
    /// eg. assets/icon.png
    pub(crate) icon: Option<Vec<String>>,
    /// eg. Extra assets like "img.png"
    pub(crate) resources: Option<Vec<String>>,
    /// eg. DioxusLabs
    pub(crate) copyright: Option<String>,
    /// eg. "Social Media"
    pub(crate) category: Option<String>,
    /// eg. "A great social media app"
    pub(crate) short_description: Option<String>,
    /// eg. "A social media app that makes people love app development"
    pub(crate) long_description: Option<String>,
    /// eg. extra binaries (like tools) to include in the final app
    pub(crate) external_bin: Option<Vec<String>>,
    /// Additional debian-only settings (see below)
    pub(crate) deb: Option<DebianSettings>,
    /// Additional macos settings (see below)
    pub(crate) macos: Option<MacOsSettings>,
    /// Additional windows settings (see below)
    pub(crate) windows: Option<WindowsSettings>,
}

#[derive(Debug, Clone, Serialize, Deserialize, Default)]
pub(crate) struct DebianSettings {
    /// OS-specific settings:
    /// the list of debian dependencies.
    pub depends: Option<Vec<String>>,
    /// the list of dependencies the package provides.
    pub provides: Option<Vec<String>>,
    /// the list of package conflicts.
    pub conflicts: Option<Vec<String>>,
    /// the list of package replaces.
    pub replaces: Option<Vec<String>>,
    /// List of custom files to add to the deb package.
}
```

```

    /// Maps the path on the debian package to the path of the file to include
    (relative to the current working directory).
    pub files: HashMap<PathBuf, PathBuf>,
    /// Path to a custom desktop file Handlebars template.
    ///
    /// Available variables: `categories`, `comment` (optional), `exec`, `icon`
    and `name`.
    pub desktop_template: Option<PathBuf>,
    /// Define the section in Debian Control file. See :
    <https://www.debian.org/doc/debian-policy/ch-archive.html#s-subsections>
    pub section: Option<String>,
    /// Change the priority of the Debian Package. By default, it is set to
    `optional`.
    /// Recognized Priorities as of now are : `required`, `important`,
    `standard`, `optional`, `extra`
    pub priority: Option<String>,
    /// Path of the uncompressed Changelog file, to be stored at
    /usr/share/doc/package-name/changelog.gz. See
    /// <https://www.debian.org/doc/debian-policy/ch-docs.html#changelog-files-
    and-release-notes>
    pub changelog: Option<PathBuf>,
    /// Path to script that will be executed before the package is unpacked. See
    /// <https://www.debian.org/doc/debian-policy/ch-maintainerscripts.html>
    pub pre_install_script: Option<PathBuf>,
    /// Path to script that will be executed after the package is unpacked. See
    /// <https://www.debian.org/doc/debian-policy/ch-maintainerscripts.html>
    pub post_install_script: Option<PathBuf>,
    /// Path to script that will be executed before the package is removed. See
    /// <https://www.debian.org/doc/debian-policy/ch-maintainerscripts.html>
    pub pre_remove_script: Option<PathBuf>,
    /// Path to script that will be executed after the package is removed. See
    /// <https://www.debian.org/doc/debian-policy/ch-maintainerscripts.html>
    pub post_remove_script: Option<PathBuf>,
}

#[derive(Debug, Clone, Serialize, Deserialize, Default)]
pub(crate) struct WixSettings {
    pub(crate) language: Vec<(String, Option<PathBuf>)>,
    pub(crate) template: Option<PathBuf>,
    pub(crate) fragment_paths: Vec<PathBuf>,
    pub(crate) component_group_refs: Vec<String>,
    pub(crate) component_refs: Vec<String>,
    pub(crate) feature_group_refs: Vec<String>,
    pub(crate) feature_refs: Vec<String>,
    pub(crate) merge_refs: Vec<String>,
    pub(crate) skip_webview_install: bool,
    pub(crate) license: Option<PathBuf>,
    pub(crate) enable_elevated_update_task: bool,
    pub(crate) banner_path: Option<PathBuf>,
    pub(crate) dialog_image_path: Option<PathBuf>,
    pub(crate) fips_compliant: bool,
    /// MSI installer version in the format `major.minor.patch.build` (build is
    optional).
    ///
    /// Because a valid version is required for MSI installer, it will be
    derived from [`PackageSettings::version`] if this field is not set.
    ///
    /// The first field is the major version and has a maximum value of 255. The
    second field is the minor version and has a maximum value of 255.
    /// The third and fourth fields have a maximum value of 65,535.
    ///
    /// See <https://learn.microsoft.com/en-us/windows/win32/msi/productversion>
    for more info.
    pub version: Option<String>,

```

```

    /// A GUID upgrade code for MSI installer. This code **_must stay the same
    across all of your updates_**,
    /// otherwise, Windows will treat your update as a different app and your
    users will have duplicate versions of your app.
    ///
    /// By default, tauri generates this code by generating a Uuid v5 using the
    string `<productName>.exe.app.x64` in the DNS namespace.
    /// You can use Tauri's CLI to generate and print this code for you by
    running `tauri inspect wix-upgrade-code`.
    ///
    /// It is recommended that you set this value in your tauri config file to
    avoid accidental changes in your upgrade code
    /// whenever you want to change your product name.
    pub upgrade_code: Option<uuid::Uuid>,
}

#[derive(Debug, Clone, Serialize, Deserialize, Default)]
pub(crate) struct MacOSSettings {
    pub(crate) frameworks: Option<Vec<String>>,
    pub(crate) minimum_system_version: Option<String>,
    pub(crate) license: Option<String>,
    pub(crate) exception_domain: Option<String>,
    pub(crate) signing_identity: Option<String>,
    pub(crate) provider_short_name: Option<String>,
    pub(crate) entitlements: Option<String>,
    pub(crate) info_plist_path: Option<PathBuf>,
    /// List of custom files to add to the application bundle.
    /// Maps the path in the Contents directory in the app to the path of the
    file to include (relative to the current working directory).
    pub files: HashMap<PathBuf, PathBuf>,
    /// Preserve the hardened runtime version flag, see
    <https://developer.apple.com/documentation/security/hardened\_runtime>
    ///
    /// Settings this to `false` is useful when using an ad-hoc signature,
    making it less strict.
    pub hardened_runtime: bool,
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub(crate) struct WindowsSettings {
    pub(crate) digest_algorithm: Option<String>,
    pub(crate) certificate_thumbprint: Option<String>,
    pub(crate) timestamp_url: Option<String>,
    pub(crate) tsp: bool,
    pub(crate) wix: Option<WixSettings>,
    pub(crate) icon_path: Option<PathBuf>,
    pub(crate) webview_install_mode: WebviewInstallMode,
    pub(crate) webview_fixed_runtime_path: Option<PathBuf>,
    pub(crate) allow_downgrades: bool,
    pub(crate) nsis: Option<NsisSettings>,
    /// Specify a custom command to sign the binaries.
    /// This command needs to have a `%1` in it which is just a placeholder for
    the binary path,
    /// which we will detect and replace before calling the command.
    ///
    /// Example:
    /// ```text
    /// sign-cli --arg1 --arg2 %1
    /// ```
    ///
    /// By Default we use `signtool.exe` which can be found only on Windows so
    /// if you are on another platform and want to cross-compile and sign you
    will
    /// need to use another tool like `osslsigncode`.

```

```

    pub sign_command: Option<CustomSignCommandSettings>,
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub(crate) struct NsisSettings {
    pub(crate) template: Option<PathBuf>,
    pub(crate) license: Option<PathBuf>,
    pub(crate) header_image: Option<PathBuf>,
    pub(crate) sidebar_image: Option<PathBuf>,
    pub(crate) installer_icon: Option<PathBuf>,
    pub(crate) install_mode: NSISInstallerMode,
    pub(crate) languages: Option<Vec<String>>,
    pub(crate) custom_language_files: Option<HashMap<String, PathBuf>>,
    pub(crate) display_language_selector: bool,
    pub(crate) start_menu_folder: Option<String>,
    pub(crate) installer_hooks: Option<PathBuf>,
    /// Try to ensure that the WebView2 version is equal to or newer than this
    version,
    /// if the user's WebView2 is older than this version,
    /// the installer will try to trigger a WebView2 update.
    pub minimum_webview2_version: Option<String>,
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub(crate) enum NSISInstallerMode {
    CurrentUser,
    PerMachine,
    Both,
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub(crate) enum WebviewInstallMode {
    Skip,
    DownloadBootstrapper { silent: bool },
    EmbedBootstrapper { silent: bool },
    OfflineInstaller { silent: bool },
    FixedRuntime { path: PathBuf },
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct CustomSignCommandSettings {
    /// The command to run to sign the binary.
    pub cmd: String,
    /// The arguments to pass to the command.
    ///
    /// "%1" will be replaced with the path to the binary to be signed.
    pub args: Vec<String>,
}

#[derive(Clone, Copy, Debug)]
pub(crate) enum PackageType {
    /// "macos"
    MacOSBundle,
    /// "ios"
    IosBundle,
    /// "msi"
    WindowsMsi,
    /// "nsis"
    Nsis,
    /// "deb"
    Deb,
    /// "rpm"
    Rpm,
}

```

```
    /// "appimage"  
    AppImage,  
    /// "dmg"  
    Dmg,  
    /// "updater"  
    Updater,  
}
```

Testing

When building application or libraries with Dioxus, you may want to include some tests to check the behavior of parts of your application. This guide will teach you how to test different parts of your Dioxus application.

Component Testing

You can use a combination of [pretty-assertions](#) and [dioxus-ssr](#) to check that two snippets of rsx are equal:

src/component_test.rs

```
use futures::FutureExt;  
use std::{cell::RefCell, sync::Arc};  
  
use dioxus::prelude::*;  
  
#[test]  
fn test() {  
    assert_rsx_eq(  
        rsx! {  
            div { "Hello world" }  
            div { "Hello world" }  
        },  
        rsx! {  
            for _ in 0..2 {  
                div { "Hello world" }  
            }  
        },  
    )  
}  
  
fn assert_rsx_eq(first: Element, second: Element) {  
    let first = dioxus_ssr::render_element(first);  
    let second = dioxus_ssr::render_element(second);  
    pretty_assertions::assert_str_eq!(first, second);  
}
```

Hook Testing

When creating libraries around Dioxus, it can be helpful to make tests for your [custom hooks](#).

Dioxus does not currently have a full hook testing library, but you can build a bespoke testing framework by manually driving the virtual dom.

src/hook_test.rs

```
use futures::FutureExt;
```

```

use std::{cell::RefCell, rc::Rc, sync::Arc, thread::Scope};

use dioxus::{dioxus_core::NoOpMutations, prelude::*};

#[test]
fn test() {
    test_hook(
        || use_signal(|| 0),
        |mut value, mut proxy| match proxy.generation {
            0 => {
                value.set(1);
            }
            1 => {
                assert_eq!(*value.read(), 1);
                value.set(2);
            }
            2 => {
                proxy.rerun();
            }
            3 => {}
            _ => todo!(),
        },
        |proxy| assert_eq!(proxy.generation, 4),
    );
}

fn test_hook<V: 'static>(
    initialize: impl FnMut() -> V + 'static,
    check: impl FnMut(V, MockProxy) + 'static,
    mut final_check: impl FnMut(MockProxy) + 'static,
) {
    #[derive(Props)]
    struct MockAppComponent<I: 'static, C: 'static> {
        hook: Rc<RefCell<I>>,
        check: Rc<RefCell<C>>,
    }

    impl<I, C> PartialEq for MockAppComponent<I, C> {
        fn eq(&self, _: &Self) -> bool {
            true
        }
    }

    impl<I, C> Clone for MockAppComponent<I, C> {
        fn clone(&self) -> Self {
            Self {
                hook: self.hook.clone(),
                check: self.check.clone(),
            }
        }
    }

    fn mock_app<I: FnMut() -> V, C: FnMut(V, MockProxy), V>(
        props: MockAppComponent<I, C>,
    ) -> Element {
        let value = props.hook.borrow_mut()();

        props.check.borrow_mut()(value, MockProxy::new());

        rsx! { div {} }
    }

    let mut vdom = VirtualDom::new_with_props(
        mock_app,

```

```

MockAppComponent {
    hook: Rc::new(RefCell::new(initialize)),
    check: Rc::new(RefCell::new(check)),
},
);

vdom.rebuild_in_place();

while vdom.wait_for_work().now_or_never().is_some() {
    vdom.render_immediate(&mut NoOpMutations);
}

vdom.in_runtime(|| {
    ScopeId::ROOT.in_runtime(|| {
        final_check(MockProxy::new());
    })
})
}

struct MockProxy {
    rerender: Arc<dyn Fn()>,
    pub generation: usize,
}

impl MockProxy {
    fn new() -> Self {
        let generation = generation();
        let rerender = schedule_update();

        Self {
            rerender,
            generation,
        }
    }

    pub fn rerun(&mut self) {
        (self.rerender)();
    }
}

```

End to End Testing

You can use [Playwright](#) to create end to end tests for your dioxus application.

In your `playwright.config.js`, you will need to run `cargo run` or `dx serve` instead of the default build command. Here is a snippet from the end to end web example:

```

//...
webServer: [
    {
        cwd: path.join(process.cwd(), 'playwright-tests', 'web'),
        command: 'dx serve',
        port: 8080,
        timeout: 10 * 60 * 1000,
        reuseExistingServer: !process.env.CI,
        stdout: "pipe",
    },
],

```

- [Web example](#)
- [Liveview example](#)

- [Fullstack example](#)

Tailwind

You can style your Dioxus application with whatever CSS framework you choose, or just write vanilla CSS.

One popular option for styling your Dioxus application is [Tailwind](#). Tailwind allows you to style your elements with CSS utility classes. This guide will show you how to setup tailwind CSS with your Dioxus application.

Setup

1. Install the Dioxus CLI:

```
cargo install dioxus-cli
```

1. Install npm: <https://docs.npmjs.com/downloading-and-installing-node-js-and-npm>
2. Install the tailwind css cli: <https://tailwindcss.com/docs/installation>
3. Initialize the tailwind css project:

```
npx tailwindcss init
```

This should create a `tailwind.config.js` file in the root of the project.

1. Edit the `tailwind.config.js` file to include rust files:

```
module.exports = {
  mode: "all",
  content: [
    // include all rust, html and css files in the src directory
    "./src/**/*.{rs,html,css}",
    // include all html files in the output (dist) directory
    "./dist/**/*.html",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

1. Create a `input.css` file in the root of your project with the following content:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

1. Create a link to the `tailwind.css` file using `manganis` somewhere in your rust code:

`src/tailwind.rs`

```
use dioxus::prelude::*;

#[component]
fn app() -> Element {
  rsx! {
```

```
// The Stylesheet component inserts a style link into the head of the
document
document::Stylesheet {
  // Urls are relative to your Cargo.toml file
  href: asset!("/assets/tailwind.css")
}
}
```

Bonus Steps

1. Install the tailwind css vs code extension
2. Go to the settings for the extension and find the experimental regex support section. Edit the setting.json file to look like this:

```
"tailwindCSS.experimental.classRegex": ["class: \"(.*)\""],
"tailwindCSS.includeLanguages": {
  "rust": "html"
},
```

Development

- Run the following command in the root of the project to start the tailwind css compiler:

```
npx tailwindcss -i ./input.css -o ./assets/tailwind.css --watch
```

Web

- Run the following command in the root of the project to start the dioxus dev server:

```
dx serve
```

- Open the browser to <http://localhost:8080>.

Desktop

- Launch the dioxus desktop app:

```
dx serve --platform desktop
```

Optimizing

Note: This is written primarily for the web, but the main optimizations will work on other platforms too.

You might have noticed that Dioxus binaries are pretty big. The WASM binary of a [TodoMVC app](#) weighs in at 2.36mb! Don't worry; we can get it down to a much more manageable 234kb. This will get obviously lower over time. With nightly features, you can even reduce the binary size of a hello world app to less than 100kb!

We will also discuss ways to optimize your app for increased speed.

However, certain optimizations will sacrifice speed for decreased binary size or the other way around. That's what you need to figure out yourself. Does your app perform performance-intensive

tasks, such as graphical processing or tons of DOM manipulations? You could go for increased speed. In most cases, though, decreased binary size is the better choice, especially because Dioxus WASM binaries are quite large.

To test binary sizes, we will use [this](#) repository as a sample app. The `no-optimizations` package will serve as the base, which weighs 2.36mb as of right now.

Additional resources:

- [WASM book - Shrinking .wasm code size](#)
- [min-sized-rust](#)

Building in release mode

This is the best way to optimize. In fact, the 2.36mb figure at the start of the guide is with release mode. In debug mode, it's actually a whopping 32mb! It also increases the speed of your app.

We can use the `--release` flag to create an optimized build of our application which will be both faster and smaller:

```
dx build --release
```

UPX

If you're not targeting web, you can use the [UPX](#) CLI tool to compress your executables.

Setup:

- Download a [release](#) and extract the directory inside to a sensible location.
- Add the executable located in the directory to your path variable.

You can run `upx --help` to get the CLI options, but you should also view `upx-doc.html` for more detailed information. It's included in the extracted directory.

An example command might be: `upx --best -o target/release/compressed.exe target/release/your-executable.exe`.

Build configuration

Note: Settings defined in `.cargo/config.toml` will override settings in `Cargo.toml`.

Other than the `--release` flag, this is the easiest way to optimize your projects, and also the most effective way, at least in terms of reducing binary size.

Stable

This configuration is 100% stable and decreases the binary size from 2.36mb to 310kb. Add this to your `.cargo/config.toml`:

```
[profile.release]
opt-level = "z"
debug = false
lto = true
codegen-units = 1
```

```
panic = "abort"
strip = true
incremental = false
```

Links to the documentation of each value:

- [opt-level](#)
- [debug](#)
- [lto](#)
- [codegen-units](#)
- [panic](#)
- [strip](#)
- [incremental](#)

Unstable

This configuration contains some unstable features, but it should work just fine. It decreases the binary size from 310kb to 234kb. Add this to your `.cargo/config.toml`:

```
[unstable]
build-std = ["std", "panic_abort", "core", "alloc"]
build-std-features = ["panic_immediate_abort"]

[build]
rustflags = [
    "-C lto",
    "-Z virtual-function-elimination",
    "-Z location-detail=none"
]

# Same as in the Stable section
[profile.release]
opt-level = "z"
debug = false
lto = true
codegen-units = 1
panic = "abort"
strip = true
incremental = false
```

Note: The omitted space in each flag (e.g., `-C<no space here>lto`) is intentional. It is not a typo.

The values in `[profile.release]` are documented in the [Stable](#) section. Links to the documentation of each value:

- [\[build.rustflags\]](#)
- [-C lto](#)
- [-Z virtual-function-elimination](#)
- [-Z location-detail](#)

wasm-opt

Note: In the future, `wasm-opt` will be supported natively through the [Dioxus CLI](#).

`wasm-opt` is a tool from the [binaryen](#) library that optimizes your WASM files. To use it, install a [binaryen release](#) and run this command from the package directory:

```
wasm-opt dist/assets/dioxus/APP_NAME_bg.wasm -o
dist/assets/dioxus/APP_NAME_bg.wasm -Oz
```

The `-Oz` flag specifies that `wasm-opt` should optimize for size. For speed, use `-O4`.

Improving Dioxus code

Let's talk about how you can improve your Dioxus code to be more performant.

It's important to minimize the number of dynamic parts in your `rsx`, like conditional rendering. When Dioxus is rendering your component, it will skip parts that are the same as the last render. That means that if you keep dynamic rendering to a minimum, your app will speed up, and quite a bit if it's not just hello world. To see an example of this, check out [Dynamic Rendering](#).

Also check out [Anti-patterns](#) for patterns that you should avoid. Obviously, not all of them are just about performance, but some of them are.

Optimizing the size of assets

Assets can be a significant part of your app's size. Dioxus includes alpha support for first party [assets](#). Any assets you include with the `asset!` macro will be optimized for production in release builds.

How to Upgrade to Dioxus 0.6

This guide will outline the API changes between the `0.5` and `0.6` releases. The `0.6` release contains a breaking changes to:

- The `Element` type
- Prevent default
- Assets with Manganis
- `dioxus_logger` integration with `dioxus`
- The `launch` function
- The `eval` function
- The `dioxus-fullstack` crate
- The router crate
- The `derive(Props)` macro
- The `dioxus-core` crate
- Custom renderer API
- Global state management

Element

The `element` type has changed from `Option<VNode>` to `Result<VNode, RenderError>`. This makes it possible to bubble up errors while rendering with the `?` operator, but it does remove

the ability to return `None` from a component. Instead of returning `None`, you can return `VNode::empty()` or an empty `rsx!` macro.

Dioxus 0.5:

```
use dioxus::prelude::*;

fn app() -> Element {
    let number = use_signal(|| -1);

    if number() < 0 {
        // ✗ In dioxus 0.6, the element type is a result, so None values cannot
        be returned directly
        return None;
    }

    rsx! {
        "Positive number: {number}"
    }
}
```

Dioxus 0.6:

src/migration.rs

```
use dioxus::prelude::*;

fn app() -> Element {
    let number = use_signal(|| -1);

    if number() < 0 {
        // ✓ You can return VNode::empty() instead
        return VNode::empty();
    }
    if number() < 0 {
        // ✓ Or an empty rsx! macro
        return rsx! {};
    }

    rsx! {
        "Positive number: {number}"
    }
}
```

Prevent Default

Dioxus 0.1-0.5 used the `prevent_default` attribute to prevent default behavior of event handlers for every event. Dioxus 0.6 introduces more fine-grained control over preventing default behavior with the `prevent_default` function on the event type. Instead of setting the `prevent_default` attribute for all events you want to prevent, you can create event handlers that call `event.prevent_default()`.

Dioxus 0.5:

src/migration.rs

```
use dioxus::prelude::*;

fn app() -> Element {
```

```

    rsx! {
      a {
        href: "https://dioxuslabs.com",
        // ✗ The prevent default attribute is deprecated in dioxus 0.6
        prevent_default: "onclick",
        "Don't navigate to dioxuslabs.com"
      }
    }
  }
}

```

Dioxus 0.6:

src/migration.rs

```

use dioxus::prelude::*;

fn app() -> Element {
  rsx! {
    a {
      href: "https://dioxuslabs.com",
      // ✓ Instead, you can call event.prevent_default() inside the event
handler
      onclick: move |event| event.prevent_default(),
      "Don't navigate to dioxuslabs.com"
    }
  }
}

```

Note: Since event handlers run on the server in Liveview, events cannot be prevented quickly inside the event handler. Because of this, the new `prevent_default` method does not prevent default behavior in Liveview.

Instead you can use javascript inside the `onclick` handler to prevent default behavior.

src/migration.rs

```

use dioxus::prelude::*;

fn app() -> Element {
  rsx! {
    a {
      href: "https://dioxuslabs.com",
      // ✓ In liveview, you can use javascript to prevent
default behavior
      "onclick": "event.preventDefault()",
      "Don't navigate to dioxuslabs.com"
    }
  }
}

```

Assets

The syntax of the `asset!` macro has changed in Dioxus 0.6. Instead of accepting a single argument with both the path and the configuration for the asset, you can now pass in the path as the first argument and the configuration as a optional second argument.

The path the `asset!` macro accepts has also changed. Previously, the macro used to accept absolute and relative paths where relative paths were relative to the current crate directory. Now the macro only accepts absolute paths which are resolved relative to the root of the crate.

Dioxus 0.5:

```
use dioxus::prelude::*;

fn app() -> Element {
    rsx! {
        img {
            src: asset!(image("./assets/static/bundle.png").size(100, 100))
        }
    }
}
```

Dioxus 0.6:

src/migration.rs

```
use dioxus::prelude::*;

fn app() -> Element {
    rsx! {
        img {
            src: asset!("/assets/static/bundle.png",
ImageAssetOptions::new().with_size(ImageSize::Manual { width: 100, height: 100
})))
        }
    }
}
```

Logging

Dioxus 0.6 brings the `dioxus - logger` crate directly into dioxus itself.

Previously, you needed to add `dioxus - logger` to your Cargo.toml and then call its init function:

```
// cargo.toml:
// dioxus-logger = "0.5"

use dioxus::prelude::*;
use tracing::Level;

fn main() {
    // Init logger
    dioxus_logger::init(Level::INFO).expect("failed to init logger");

    // Dioxus launch code
    dioxus::launch(app)
}
```

Now, in Dioxus 0.6, the logger is implicit with `launch`. Simply call `launch` and the logger is initialized to a default log level. In development mode, the `Debug` tracing level is set, and in release only the `Info` level is set.

```
use dioxus::prelude::*;

fn main() {
```

```
dioxus::launch(app);  
}
```

If you still need to set the level manually or configure a custom subscriber, do that before `launch`. We expose the `initialize_default` function in case you need additional logging before your `launch` call:

```
use dioxus::prelude::*;  
  
fn main() {  
    dioxus::logger::initialize_default();  
  
    tracing::info!("Logs received!");  
  
    dioxus::launch(app);  
}
```

Launch

The `launch` function was removed from the prelude. You must now import the launch method from `dioxus` or use it by its full path:

```
use dioxus::prelude::*;  
  
fn main() {  
    // ✗ launch(app);  
    dioxus::launch(app); // ✓  
}
```

See <https://github.com/DioxusLabs/dioxus/pull/2967> for more details.

Eval

- `eval` was moved from the prelude to the `document` module. You must now call it with `document::eval` instead of `eval`:

```
use dioxus::prelude::*;  
  
fn app() -> Element {  
    // ✗ use_effect(|| eval("console.log(1)"));  
    use_effect(|| document::eval("console.log(1)")); // ✓  
  
    rsx! {}  
}
```

- The `eval` feature flag was removed from the `dioxus-html` crate and the functionality of `EvalProvider` was moved to the new `dioxus-document` crate. Custom renderers must now provide a `Rc<dyn Document>` context to the application to make `eval` and head elements work correctly. See <https://github.com/DioxusLabs/dioxus/pull/2635> for more details.
- `Eval::recv` and `Eval::join` now returns any value that implements `DeserializeOwned` instead of `serde_json::Value`. `Eval::send` now accepts any value that implements `Serialize`. See <https://github.com/DioxusLabs/dioxus/pull/3035> for more details

Fullstack

- The feature `dioxus/axum` was renamed to `dioxus/server`

```
[features]
default = []
# ✕ server = ["dioxus/axum"]
server = ["dioxus/server"] # ✓
web = ["dioxus/web"]
```

See <https://github.com/DioxusLabs/dioxus/pull/3186> for more details

- The `fullstack::Config` item was removed. You can now pass the platform configs into the `LaunchBuilder` directly. For example, if you want to set the rootname on each platform, you can set the root name in each config:

```
LaunchBuilder::new()
// Only set the server config if the server feature is enabled
.with_cfg(server_only! {
    ServeConfigBuilder::default().root_id("app")
})
// You also need to set the root id in your web config
.with_cfg(web! {
    dioxus::web::Config::default().rootname("app")
})
// And desktop config
.with_cfg(desktop! {
    dioxus::desktop::Config::default().with_root_name("app")
})
.launch(app);
```

See <https://github.com/DioxusLabs/dioxus/pull/2967> for more details.

- The `dioxus-cli` now proxies fullstack applications at a port behind a reverse proxy. If you have a custom axum server, you must serve your application at the port returned by `dioxus_cli_config::server_port` and the address returned by `dioxus_cli_config::server_ip` or the complete address returned by `dioxus_cli_config::fullstack_address_or_localhost` during development:

```
#[cfg(feature = "server")]
#[tokio::main]
async fn main() {
    // Get the address the server should run on. If the CLI is running, the CLI
    proxies fullstack into the main address
    // and we use the generated address the CLI gives us
    let address = dioxus_cli_config::fullstack_address_or_localhost();

    // Launch the fullstack application on the address the CLI is proxying
    let router = axum::Router::new()
        .serve_dioxus_application(ServeConfigBuilder::default(), App);

    let router = router.into_make_service();
    let listener = tokio::net::TcpListener::bind(address).await.unwrap();
    axum::serve(listener, router).await.unwrap();
}
```

See <https://github.com/DioxusLabs/dioxus/pull/2258> for more details.

- `serve_dioxus_application` was changed to accept a component directly instead of a virtual dom factory. See <https://github.com/DioxusLabs/dioxus/pull/2515> for more details.
- `register_server_fns` was renamed to `register_server_functions`. See <https://github.com/DioxusLabs/dioxus/pull/2515> for more details.
- `RenderHandleState::new` accepts a new `ServeConfig` argument. See <https://github.com/DioxusLabs/dioxus/pull/2862> for more details.
- `ServeConfigBuilder::build` returns a result. It may fail during desktop builds if no `index.html` file is found. This error is fine to ignore in desktop builds. You can pass the builder directly to `serve_dioxus_application` to only serve the `index.html` file if it exists. See <https://github.com/DioxusLabs/dioxus/pull/2862> for more details.
- `dioxus_fullstack::Config::addr` was removed. You can now export the `PORT` and `IP` environment variables to set the address the `launch` method uses for the server.

Router

- The `Routeable` derive macro no longer accepts fields that are not present in the `route("/route")` if the web feature is enabled. See <https://github.com/DioxusLabs/dioxus/pull/2159> for more details.
- The `ToRouteSegments` trait in the router was changed from accepting `self` to accepting `&self`. This means you can now implement it for `T` directly instead of `&T`. See <https://github.com/DioxusLabs/dioxus/pull/2283> for more details.

derive(Props)

- `#[props(into)]` is ignore on any `String` props. `String` props already accept `impl ToString` which is implemented for many of the same types, but if you implement `Into<String>` for a specific type, your code may require some changes. See <https://github.com/DioxusLabs/dioxus/pull/2501> for more details
- Properties that start with an uppercase letter are no longer accepted. This allows us to autocomplete Components. See <https://github.com/DioxusLabs/dioxus/pull/2652> for more details.

State Management

- `use_coroutine` now accepts `impl FnMut` instead of `impl FnOnce`. This was required to support restarting the coroutine without rerunning the component. See <https://github.com/DioxusLabs/dioxus/pull/3005> for more details.
- `Signal::global_memo` now requires `T: PartialEq` just like `use_memo`. See <https://github.com/DioxusLabs/dioxus/pull/2851> for more details.
- `GlobalMemo<T>` is now a trait alias for `Global<Memo<T>, T>` and `GlobalSignal<T>` is now a trait alias for `Global<Signal<T>, T>`. To get the underlying `Memo` or `Signal`, you can now use the `resolve` method instead of `signal` or `memo`. See <https://github.com/DioxusLabs/dioxus/pull/2851> for more details.

- The `Readable` trait in `dioxus` signals now requires a `try_peek_unchecked` method instead of `peek_unchecked`. See <https://github.com/DioxusLabs/dioxus/pull/2714> for more details.
- The `check_generation` feature flag was removed from the `generational-box` crate. See <https://github.com/DioxusLabs/dioxus/pull/2638> for more details.

Core changes

- The `Template::name` field was removed. See <https://github.com/DioxusLabs/dioxus/pull/2799> for more details.
- `Properties::into_vcomponent` now accepts only the `render_fn` instead of the `render_fn` and `component_name`. This change fixes the name of re-exported components. Fixes <https://github.com/DioxusLabs/dioxus/pull/2744>
- The field `VNode::template` is now `Template` instead of `Cell<Template>`. See <https://github.com/DioxusLabs/dioxus/pull/2705> for more details
- `Mutations::santize` was renamed to `Mutations::sanitize`. See <https://github.com/DioxusLabs/dioxus/pull/2653> for more details.
- The variant `AttributeValue::Any` now contains `Rc<dyn AnyValue>` instead of `Box<dyn AnyValue>` to make `AttributeValue Clone`. See <https://github.com/DioxusLabs/dioxus/pull/2705> for more details

Custom Renderers

If you are building a custom renderer, there were some breaking changes to hot reloading and rsx that you should be aware of:

- The CLI hot reloading format changed significantly. Custom renderers must switch from `dioxus-hot-reload` to `dioxus_devtools`. Renderers can connect to the hot reloading engine with the `[ connect ]` (https://docs.rs/dioxus-devtools/0.6.0/dioxus_devtools/fn.connect.html) function. See <https://github.com/DioxusLabs/dioxus/pull/2258> for more details.
- The format of custom elements was changed to improve autocomplete. The `dioxus_elements` namespace must now contain each element as a module with a `TAG_NAME` and `NAME_SPACE` constant inside that module. Each attribute should be another constant in that module. The top-level `dioxus_elements` module should contain a `completions` module with a `CompleteWithBraces` enum that re-exports each element the namespace supports for braces autocomplete. See <https://github.com/DioxusLabs/dioxus/pull/2421> for more details.
- The format for custom event handlers changed include `eventname::call_with_explicit_closure` to provide better type inference for inline closures. See <https://github.com/DioxusLabs/dioxus/pull/2437> for more details

If you are also using `dioxus-html`, there are a few more breaking changes:

- A `file_size` method was added to the `FileEngine` trait. Any custom renderers must implement this method. See <https://github.com/DioxusLabs/dioxus/pull/2323/files> for more details.

- `HtmlEventConverter` has a new `convert_resize_data` method which must be implemented by any custom renderers that use `dioxus-html`. See <https://github.com/DioxusLabs/dioxus/pull/2479> for more details
- The web and native features were removed from the `dioxus-html` crate. See <https://github.com/DioxusLabs/dioxus/pull/3006> for more details.
- `dioxus_html::AttributeDiscription` was renamed to `dioxus_html::AttributeDescription`. See <https://github.com/DioxusLabs/dioxus/pull/2653> for more details.

Minor Breaking Changes

There were several more minor breaking changes in Dioxus 0.6:

- Many implicit features from dioxus crates were removed. These features were automatically generated by cargo and generally not functional. See <https://github.com/DioxusLabs/dioxus/pull/2512> for more details.
- `dioxus_autofmt::write_block_out` accepts `&CallBody` instead of `CallBody`. See <https://github.com/DioxusLabs/dioxus/pull/2573> for more details.
- The `panic_hook` feature which provides a console panic message for wasm panics was moved from the `dioxus-web` crate to the `dioxus-logger` crate. The handler is still provided by default. See <https://github.com/DioxusLabs/dioxus/pull/3302> for more details.

Reference

This portion of the Dioxus Docs contains specific details about features that provide more depth than what is necessary for most reading.

- [RSX](#)
- [Components](#)
- [Props](#)
- [Event Handlers](#)
- [Hooks](#)
- [User Input](#)
- [Context](#)
- [Dynamic Rendering](#)
- [Routing](#)
- [Resource](#)
- [UseCoroutine](#)
- [Spawn](#)

Dioxus Hot-Reloading Reference

The Dioxus Hot-Reload is very powerful. When used properly, it is by-far the fastest tool to build apps with Rust.

Dioxus 0.5 featured a slightly limited form of hot-reloading while Dioxus 0.6 drastically improved it.

Currently Dioxus cannot hot-reload *Rust* code, only RSX markup. Usually, modifying Rust code requires a full rebuild.

We provide this text guide as a resource for the details of hot-reloading. This guide also has an accompanying video as well:

What can be hot-reloaded?

Within RSX, all elements and their properties can be hot-reloaded.

```
rsx! {  
  div {}  
}
```

What causes a full-rebuild?

Describing the UI

Dioxus is a *declarative* framework. This means that instead of telling Dioxus what to do (e.g. to "create an element" or "set the color to red") we simply *declare* what we want the UI to look like using RSX.

You have already seen a simple example of RSX syntax in the "hello world" application:

src/hello_world_desktop.rs

```
// define a component that renders a div with the text "Hello, world!"  
fn App() -> Element {  
  rsx! {  
    div { "Hello, world!" }  
  }  
}
```

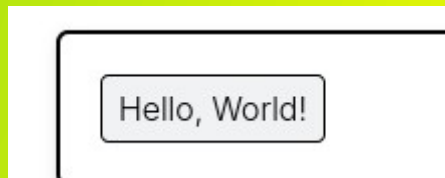
Here, we use the `rsx!` macro to *declare* that we want a `div` element, containing the text "Hello, world!". Dioxus takes the RSX and constructs a UI from it.

RSX Features

RSX is very similar to HTML in that it describes elements with attributes and children. Here's an empty `button` element in RSX, as well as the resulting HTML:

src/rsx_overview.rs

```
rsx! {  
  button {  
    // attributes / listeners  
    // children  
    "Hello, World!"  
  }  
}
```



Attributes

Attributes (and [event handlers](#)) modify the behavior or appearance of the element they are attached to. They are specified inside the `{ }` brackets, using the `name: value` syntax. You can provide the value as a literal in the RSX:

src/rsx_overview.rs

```
rsx! {  
  img {  
    src: "https://avatars.githubusercontent.com/u/79236386?s=200&v=4",  
    class: "primary_button",  
    width: "100px",  
  }  
}
```



Some attributes, such as the `type` attribute for `input` elements won't work on their own in Rust. This is because `type` is a reserved Rust keyword. To get around this, Dioxus uses the `r#` specifier:

src/rsx_overview.rs

```
rsx! {  
  input { r#type: "text", color: "red" }  
}
```

Note: All attributes defined in `dioxus-html` follow the `snake_case` naming convention. They transform their `snake_case` names to HTML's `camelCase` attributes.

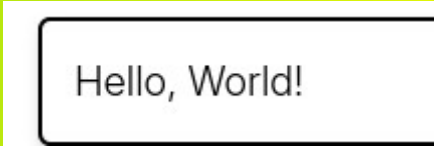
Note: Styles can be used directly outside of the `style:` attribute. In the above example, `color: "red"` is turned into `style="color: red"`.

Conditional Attributes

You can also conditionally include attributes by using an `if` statement without an `else` branch. This is useful for adding an attribute only if a certain condition is met:

src/rsx_overview.rs

```
let large_font = true;  
rsx! {  
  div { class: if large_font { "text-xl" }, "Hello, World!" }  
}
```



Hello, World!

Custom Attributes

Dioxus has a pre-configured set of attributes that you can use. RSX is validated at compile time to make sure you didn't specify an invalid attribute. If you want to override this behavior with a custom attribute name, specify the attribute in quotes:

src/rsx_overview.rs

```
rsx! {  
  div { "style": "width: 20px; height: 20px; background-color: red;" }  
}
```



Special Attributes

While most attributes are simply passed on to the HTML, some have special behaviors.

The HTML Escape Hatch

If you're working with pre-rendered assets, output from templates, or output from a JS library, then you might want to pass HTML directly instead of going through Dioxus. In these instances, reach for `dangerous_inner_html`.

For example, shipping a markdown-to-Dioxus converter might significantly bloat your final application size. Instead, you'll want to pre-render your markdown to HTML and then include the HTML directly in your output. We use this approach for the [Dioxus homepage](#):

src/dangerous_inner_html.rs

```
// this should come from a trusted source  
let contents = "live <b>dangerously</b>";  
  
rsx! {  
  div { dangerous_inner_html: "{contents}" }  
}
```



live **dangerously**

Note! This attribute is called "dangerous_inner_html" because it is **dangerous** to pass it data you don't trust. If you're not careful, you can easily expose [cross-site scripting \(XSS\)](#) attacks to your users.

If you're handling untrusted input, make sure to sanitize your HTML before passing it into `dangerous_inner_html` – or just pass it to a Text Element to escape any HTML tags.

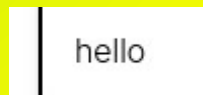
Boolean Attributes

Most attributes, when rendered, will be rendered exactly as the input you provided. However, some attributes are considered "boolean" attributes and just their presence determines whether they affect the output. For these attributes, a provided value of `false` will cause them to be removed from the target element.

So this JSX wouldn't actually render the `hidden` attribute:

src/boolean_attribute.rs

```
rsx! {  
  div { hidden: false, "hello" }  
}
```



Not all attributes work like this however. *Only the following attributes* have this behavior:

- `allowfullscreen`
- `allowpaymentrequest`
- `async`
- `autofocus`
- `autoplay`
- `checked`
- `controls`
- `default`
- `defer`
- `disabled`
- `formnovalidate`
- `hidden`
- `ismap`
- `itemscope`
- `loop`
- `multiple`
- `muted`
- `nomodule`
- `novalidate`
- `open`
- `playsinline`
- `readonly`
- `required`
- `reversed`
- `selected`

- `trueSpeed`

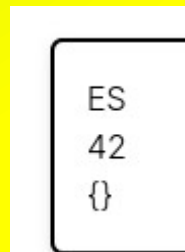
For any other attributes, a value of `false` will be sent directly to the DOM.

Interpolation

Similarly to how you can [format](#) Rust strings, you can also interpolate in JSX text. Use `{variable}` to Display the value of a variable in a string, or `{variable:?}` to use the Debug representation:

src/rsx_overview.rs

```
let coordinates = (42, 0);
let country = "es";
rsx! {
  div {
    class: "country-{{country}}",
    left: "{coordinates.0:?}" ,
    top: "{coordinates.1:?}" ,
    // arbitrary expressions are allowed,
    // as long as they don't contain `{}`
    div { "{country.to_uppercase()}" }
    div { "{7*6}" }
    // {} can be escaped with {{}}
    div { "{{{}}}" }
  }
}
```

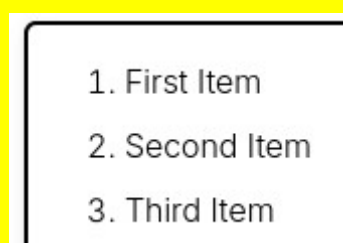


Children

To add children to an element, put them inside the `{}` brackets after all attributes and listeners in the element. They can be other elements, text, or [components](#). For example, you could have an `ol` (ordered list) element, containing 3 `li` (list item) elements, each of which contains some text:

src/rsx_overview.rs

```
rsx! {
  ol {
    li { "First Item" }
    li { "Second Item" }
    li { "Third Item" }
  }
}
```

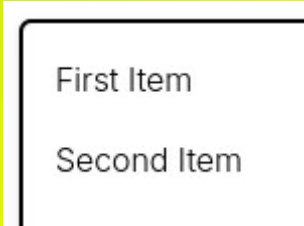


Fragments

You can render multiple elements at the top level of `rsx!` and they will be automatically grouped.

src/rsx_overview.rs

```
rsx! {  
  p { "First Item" }  
  p { "Second Item" }  
}
```



First Item

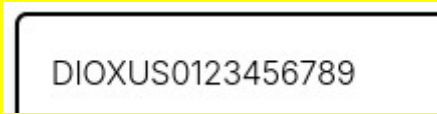
Second Item

Expressions

You can include arbitrary Rust expressions as children within RSX by surrounding your expression with `{}`s. Any expression that implements [IntoDynNode](#) can be used within `rsx`. This is useful for displaying data from an [iterator](#):

src/rsx_overview.rs

```
let text = "Dioxus";  
rsx! {  
  span {  
    {text.to_uppercase()}  
    // create a list of text from 0 to 9  
    {(0..10).map(|i| rsx! {  
      "{i}"  
    })}  
  }  
}
```



DIOXUS0123456789

Loops

In addition to iterators you can also use for loops directly within JSX:

src/rsx_overview.rs

```
rsx! {  
    // use a for loop where the body itself is JSX  
    div {  
        // create a list of text from 0 to 9  
        for i in 0..3 {  
            // NOTE: the body of the loop is JSX not a rust statement  
            div { "{i}" }  
        }  
    }  
    // iterator equivalent  
    div {  
        {(0..3).map(|i| rsx! {  
            div { "{i}" }  
        })}  
    }  
}
```

0
1
2
0
1
2

If statements

You can also use if statements without an else branch within JSX:

src/rsx_overview.rs

```
rsx! {  
    // use if statements without an else  
    if true {  
        div { "true" }  
    }  
}
```

true

Components

Just like you wouldn't want to write a complex program in a single, long, `main` function, you shouldn't build a complex UI in a single `App` function. Instead, you should break down the functionality of an app in logical parts called components.

A component is a Rust function, named in UpperCamelCase, that either takes no parameters or a properties struct and returns an `Element` describing the UI it wants to render.

`src/hello_world_desktop.rs`

```
// define a component that renders a div with the text "Hello, world!"
fn App() -> Element {
    rsx! {
        div { "Hello, world!" }
    }
}
```

You'll probably want to add `#![allow(non_snake_case)]` to the top of your crate to avoid warnings about UpperCamelCase component names

A Component is responsible for some rendering task – typically, rendering an isolated part of the user interface. For example, you could have an `About` component that renders a short description of Dioxus Labs:

`src/components.rs`

```
pub fn About() -> Element {
    rsx! {
        p {
            b { "Dioxus Labs" }
            " An Open Source project dedicated to making Rust UI wonderful."
        }
    }
}
```

Dioxus Labs An Open Source project dedicated to making Rust UI wonderful.

Then, you can render your component in another component, similarly to how elements are rendered:

`src/components.rs`

```
pub fn App() -> Element {
    rsx! {
        About {}
        About {}
    }
}
```

Dioxus Labs An Open Source project dedicated to making Rust UI wonderful.

Dioxus Labs An Open Source project dedicated to making Rust UI wonderful.

At this point, it might seem like components are nothing more than functions. However, as you learn more about the features of Dioxus, you'll see that they are actually more powerful!

Component Props

Just like you can pass arguments to a function or attributes to an element, you can pass props to a component that customize its behavior! The components we've seen so far didn't accept any props – so let's write some components that do.

derive(Props)

Component props are a single struct annotated with `#[derive(PartialEq, Clone, Props)]`. For a component to accept props, the type of its argument must be `YourPropsStruct`.

Example:

src/component_owned_props.rs

```
#[derive(PartialEq, Props, Clone)]
struct LikesProps {
    score: i32,
}

fn Likes(props: LikesProps) -> Element {
    rsx! {
        div {
            "This post has "
            b { "{props.score}" }
            " likes"
        }
    }
}
```

You can then pass prop values to the component the same way you would pass attributes to an element:

src/component_owned_props.rs

```
pub fn App() -> Element {
    rsx! { Likes { score: 42 } }
}
```

This post has **42** likes

Prop Options

The `#[derive(Props)]` macro has some features that let you customize the behavior of props.

Optional Props

You can create optional fields by using the `Option<...>` type for a field:

src/component_props_options.rs

```
#[derive(PartialEq, Clone, Props)]
struct OptionalProps {
    title: String,
    subtitle: Option<String>,
}

fn Title(props: OptionalProps) -> Element {
    rsx! {
        h1 { "{props.title}: ", {props.subtitle.unwrap_or_else(|| "No subtitle
provided".to_string())} }
    }
}
```

Then, you can choose to either provide them or not:

src/component_props_options.rs

```
Title { title: "Some Title" }
Title { title: "Some Title", subtitle: "Some Subtitle" }
// Providing an Option explicitly won't compile though:
// Title {
//     title: "Some Title",
//     subtitle: None,
// },
```

Explicitly Required Option

If you want to explicitly require an `Option`, and not an optional prop, you can annotate it with `#[props(!optional)]`:

src/component_props_options.rs

```
#[derive(PartialEq, Clone, Props)]
struct ExplicitOptionProps {
    title: String,
    #[props(!optional)]
    subtitle: Option<String>,
}

fn ExplicitOption(props: ExplicitOptionProps) -> Element {
    rsx! {
        h1 { "{props.title}: ", {props.subtitle.unwrap_or_else(|| "No subtitle
provided".to_string())} }
    }
}
```

Then, you have to explicitly pass either `Some("str")` or `None`:

src/component_props_options.rs

```
ExplicitOption { title: "Some Title", subtitle: None }
ExplicitOption { title: "Some Title", subtitle: Some("Some Title".to_string()) }
// This won't compile:
// ExplicitOption {
//     title: "Some Title",
// },
```

Default Props

You can use `#[props(default = 42)]` to make a field optional and specify its default value:

src/component_props_options.rs

```
#[derive(PartialEq, Props, Clone)]
struct DefaultProps {
    // default to 42 when not provided
    #[props(default = 42)]
    number: i64,
}

fn DefaultComponent(props: DefaultProps) -> Element {
    rx! { h1 { "{props.number}" } }
}
```

Then, similarly to optional props, you don't have to provide it:

src/component_props_options.rs

```
DefaultComponent { number: 5 }
DefaultComponent {}
```

Automatic Conversion with into

It is common for Rust functions to accept `impl Into<SomeType>` rather than just `SomeType` to support a wider range of parameters. If you want similar functionality with props, you can use `#[props(into)]`. For example, you could add it on a `String` prop – and `&str` will also be automatically accepted, as it can be converted into `String`:

src/component_props_options.rs

```
#[derive(PartialEq, Props, Clone)]
struct IntoProps {
    #[props(into)]
    string: String,
}

fn IntoComponent(props: IntoProps) -> Element {
    rx! { h1 { "{props.string}" } }
}
```

Then, you can use it so:

src/component_props_options.rs

```
IntoComponent { string: "some &str" }
```

The component macro

So far, every Component function we've seen had a corresponding ComponentProps struct to pass in props. This was quite verbose... Wouldn't it be nice to have props as simple function arguments? Then we wouldn't need to define a Props struct, and instead of typing `props.whatever`, we could just use `whatever` directly!

`component` allows you to do just that. Instead of typing the "full" version:

```
#[derive(Props, Clone, PartialEq)]
struct TitleCardProps {
    title: String,
}

fn TitleCard(props: TitleCardProps) -> Element {
    rsx!{
        h1 { "{props.title}" }
    }
}
```

...you can define a function that accepts props as arguments. Then, just annotate it with `#[component]`, and the macro will turn it into a regular Component for you:

```
#[component]
fn TitleCard(title: String) -> Element {
    rsx!{
        h1 { "{title}" }
    }
}
```

While the new Component is shorter and easier to read, this macro should not be used by library authors since you have less control over Prop documentation.

Component Children

In some cases, you may wish to create a component that acts as a container for some other content, without the component needing to know what that content is. To achieve this, create a prop of type `Element`:

`src/component_element_props.rs`

```
#[derive(PartialEq, Clone, Props)]
struct ClickableProps {
    href: String,
    body: Element,
}

fn Clickable(props: ClickableProps) -> Element {
    rsx! {
        a { href: "{props.href}", class: "fancy-button", {props.body} }
    }
}
```

Then, when rendering the component, you can pass in the output of `rsx!{ ... }`:

`src/component_element_props.rs`

```
rsx! {
```

```
Clickable {
  href: "https://www.youtube.com/watch?v=C-M2hs3sXGo",
  body: rsx! {
    "How to " i { "not" } " be seen"
  }
}
```

Warning: While it may compile, do not include the same `Element` more than once in the RSX. The resulting behavior is unspecified.

The children field

Rather than passing the RSX through a regular prop, you may wish to accept children similarly to how elements can have children. The "magic" `children` prop lets you achieve this:

src/component_children.rs

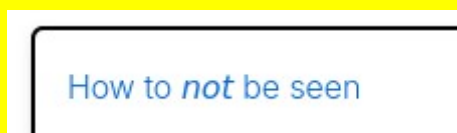
```
#[derive(PartialEq, Clone, Props)]
struct ClickableProps {
  href: String,
  children: Element,
}

fn Clickable(props: ClickableProps) -> Element {
  rsx! {
    a { href: "{props.href}", class: "fancy-button", {props.children} }
  }
}
```

This makes using the component much simpler: simply put the RSX inside the `{ }` brackets – and there is no need for a `render` call or another macro!

src/component_children.rs

```
rsx! {
  Clickable { href: "https://www.youtube.com/watch?v=C-M2hs3sXGo",
    "How to "
    i { "not" }
    " be seen"
  }
}
```



Event Handlers

Event handlers are used to respond to user actions. For example, an event handler could be triggered when the user clicks, scrolls, moves the mouse, or types a character.

Event handlers are attached to elements. For example, we usually don't care about all the clicks that happen within an app, only those on a particular button.

Event handlers are similar to regular attributes, but their name usually starts with `on-` and they accept closures as values. The closure will be called whenever the event it listens for is triggered and will be passed that event.

For example, to handle clicks on an element, we can specify an `onclick` handler:

src/event_click.rs

```
rsx! {  
  button { onclick: move |event| log::info!("Clicked! Event: {event:?}"),  
    "click me!" }  
}
```

The Event object

Event handlers receive an [Event](#) object containing information about the event. Different types of events contain different types of data. For example, mouse-related events contain [MouseData](#), which tells you things like where the mouse was clicked and what mouse buttons were used.

In the example above, this event data was logged to the terminal:

```
Clicked! Event: UiEvent { bubble_state: Cell { value: true }, data: MouseData  
{ coordinates: Coordinates { screen: (242.0, 256.0), client: (26.0, 17.0),  
element: (16.0, 7.0), page: (26.0, 17.0) }, modifiers: (empty), held_buttons:  
EnumSet(), trigger_button: Some(Primary) } }  
Clicked! Event: UiEvent { bubble_state: Cell { value: true }, data: MouseData  
{ coordinates: Coordinates { screen: (242.0, 256.0), client: (26.0, 17.0),  
element: (16.0, 7.0), page: (26.0, 17.0) }, modifiers: (empty), held_buttons:  
EnumSet(), trigger_button: Some(Primary) } }
```

To learn what the different event types for HTML provide, read the [events module docs](#).

Event propagation

Some events will trigger first on the element the event originated at upward. For example, a click event on a `button` inside a `div` would first trigger the button's event listener and then the `div`'s event listener.

For more information about event propagation see [the mdn docs on event bubbling](#)

If you want to prevent this behavior, you can call `stop_propagation()` on the event:

src/event_nested.rs

```
rsx! {  
  div { onclick: move |_event| {},  
    "outer"  
    button {  
      onclick: move |event| {  
        event.stop_propagation();  
      },  
      "inner"  
    }  
  }  
}
```

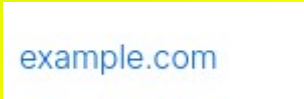
Prevent Default

Some events have a default behavior. For keyboard events, this might be entering the typed character. For mouse events, this might be selecting some text.

In some instances, might want to avoid this default behavior. For this, you can add the `prevent_default` attribute with the name of the handler whose default behavior you want to stop. This attribute can be used for multiple handlers using their name separated by spaces:

src/event_prevent_default.rs

```
rsx! {
  a {
    href: "https://example.com",
    onclick: |evt| {
      evt.prevent_default();
      log::info!("link clicked")
    },
    "example.com"
  }
}
```



Any event handlers will still be called.

Handler Props

Sometimes, you might want to make a component that accepts an event handler. A simple example would be a `FancyButton` component, which accepts an `onclick` handler:

src/event_handler_prop.rs

```
#[derive(PartialEq, Clone, Props)]
pub struct FancyButtonProps {
  onclick: EventHandler<MouseEvent>,
}

pub fn FancyButton(props: FancyButtonProps) -> Element {
  rsx! {
    button {
      class: "fancy-button",
      onclick: move |evt| props.onclick.call(evt),
      "click me pls."
    }
  }
}
```

Then, you can use it like any other handler:

src/event_handler_prop.rs

```
rsx! {
  FancyButton {
    onclick: move |event| println!("Clicked! {event:?}"),
  }
}
```

Note: just like any other attribute, you can name the handlers anything you want! Any closure you pass in will automatically be turned into an `EventHandler`.

Custom Data

Event Handlers are generic over any type, so you can pass in any data you want to them, e.g:

src/event_handler_prop.rs

```
struct ComplexData(i32);

#[derive(PartialEq, Clone, Props)]
pub struct CustomFancyButtonProps {
    onclick: EventHandler<ComplexData>,
}

pub fn CustomFancyButton(props: CustomFancyButtonProps) -> Element {
    rsx! {
        button {
            class: "fancy-button",
            onclick: move |_| props.onclick.call(ComplexData(0)),
            "click me pls."
        }
    }
}
```

Returning a value from an event handler

If you want to accept a closure like an event handler that returns a value, you can use the `Callback` type. The callback type accepts two generic arguments, `I`, the input type, and `O`, the output type. Just like `EventHandler`, `Callback` is automatically converted in props and can be easily copied into anywhere in your component:

src/event_handler_prop.rs

```
#[derive(PartialEq, Clone, Props)]
pub struct CounterProps {
    modify: Callback<u32, u32>,
}

pub fn Counter(props: CounterProps) -> Element {
    let mut count = use_signal(|| 1);

    rsx! {
        button {
            onclick: move |_| count.set(props.modify.call(count())),
            "double"
        }
        div { "count: {count}" }
    }
}
```

Hooks and component state

So far, our components have had no state like a normal Rust function. However, in a UI component, it is often useful to have stateful functionality to build user interactions. For example, you might want to track whether the user has opened a drop-down and render different things accordingly.

Hooks allow us to create state in our components. Hooks are Rust functions you call in a constant order in a component that add additional functionality to the component.

Dioxus provides many built-in hooks, but if those hooks don't fit your specific use case, you also can [create your own hook](#)

use_signal hook

`use_signal` is one of the simplest hooks.

- You provide a closure that determines the initial value: `let mut count = use_signal(|| 0);`
- `use_signal` gives you the current value, and a way to write to the value
- When the value updates, `use_signal` makes the component re-render (along with any other component that references it), and then provides you with the new value.

For example, you might have seen the counter example, in which state (a number) is tracked using the `use_signal` hook:

src/hooks_counter.rs

```
pub fn App() -> Element {  
    // count will be initialized to 0 the first time the component is rendered  
    let mut count = use_signal(|| 0);  
  
    rsx! {  
        h1 { "High-Five counter: {count}" }  
        button { onclick: move |_| count += 1, "Up high!" }  
        button { onclick: move |_| count -= 1, "Down low!" }  
    }  
}
```

High-Five counter: 1

Up high!

Down low!

Every time the component's state changes, it re-renders, and the component function is called, so you can describe what you want the new UI to look like. You don't have to worry about "changing" anything – describe what you want in terms of the state, and Dioxus will take care of the rest!

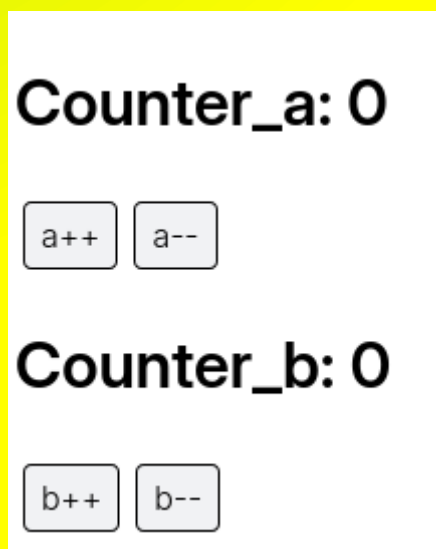
`use_signal` returns your value wrapped in a smart pointer of type [Signal](#) that is `Copy`. This is why you can both read the value and update it, even within an event handler.

You can use multiple hooks in the same component if you want:

src/hooks_counter_two_state.rs

```
pub fn App() -> Element {
    let mut count_a = use_signal(|| 0);
    let mut count_b = use_signal(|| 0);

    rsx! {
        h1 { "Counter_a: {count_a}" }
        button { onclick: move |_| count_a += 1, "a++" }
        button { onclick: move |_| count_a -= 1, "a--" }
        h1 { "Counter_b: {count_b}" }
        button { onclick: move |_| count_b += 1, "b++" }
        button { onclick: move |_| count_b -= 1, "b--" }
    }
}
```

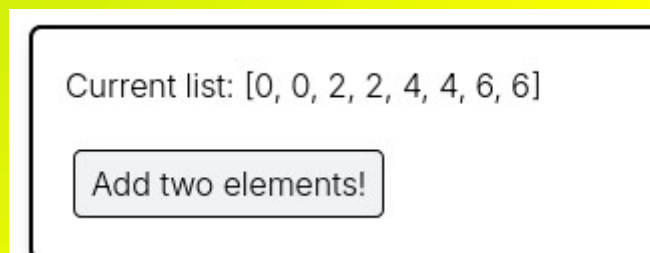


You can also use `use_signal` to store more complex state, like a `Vec`. You can read and write to the state with the `read` and `write` methods:

src/hooks_use_signal.rs

```
pub fn App() -> Element {
    let mut list = use_signal(Vec::new);

    rsx! {
        p { "Current list: {list:?}" }
        button {
            onclick: move |event| {
                let list_len = list.len();
                list.push(list_len);
                list.push(list_len);
            },
            "Add two elements!"
        }
    }
}
```



Rules of hooks

The above example might seem a bit magic since Rust functions are typically not associated with state. Dioxus allows hooks to maintain state across renders through a hidden scope that is associated with the component.

But how can Dioxus differentiate between multiple hooks in the same component? As you saw in the second example, both `use_signal` functions were called with the same parameters, so how come they can return different things when the counters are different?

src/hooks_counter_two_state.rs

```
let mut count_a = use_signal(|| 0);
let mut count_b = use_signal(|| 0);
```

This is only possible because the two hooks are always called in the same order, so Dioxus knows which is which. Because the order you call hooks matters, you must follow certain rules when using hooks:

1. Hooks may be only used in components or other hooks (we'll get to that later).
2. On every call to a component function.
3. The same hooks must be called (except in the case of early returns, as explained later in the [Error Handling chapter](#)).
4. In the same order.
5. Hook names should start with `use_` so you don't accidentally confuse them with regular functions (`use_signal()`, `use_signal()`, `use_resource()`, etc...).

These rules mean that there are certain things you can't do with hooks:

No hooks in conditionals

src/hooks_bad.rs

```
// ✗ don't call hooks in conditionals!
// We must ensure that the same hooks will be called every time
// But `if` statements only run if the conditional is true!
// So we might violate rule 2.
if you_are_happy && you_know_it {
    let something = use_signal(|| "hands");
    println!("clap your {something}")
}

// ✓ instead, *always* call use_signal
// You can put other stuff in the conditional though
let something = use_signal(|| "hands");
if you_are_happy && you_know_it {
    println!("clap your {something}")
}
```

No hooks in closures

src/hooks_bad.rs

```
// ✗ don't call hooks inside closures!
// We can't guarantee that the closure, if used, will be called in the same
// order every time
let _a = || {
    let b = use_signal(|| 0);
    b()
};

// ✓ instead, move hook `b` outside
let b = use_signal(|| 0);
let _a = || b();
```

No hooks in loops

src/hooks_bad.rs

```
// `names` is a Vec<&str>

// ✗ Do not use hooks in loops!
// In this case, if the length of the Vec changes, we break rule 2
for _name in &names {
    let is_selected = use_signal(|| false);
    println!("selected: {is_selected}");
}

// ✓ Instead, use a hashmap with use_signal
let selection_map = use_signal(HashMap:::<&str, bool>::new);

for name in &names {
    let is_selected = selection_map.read()[name];
    println!("selected: {is_selected}");
}
```

Additional resources

- [dioxus hooks API docs](#)
- [dioxus hooks source code](#)

User Input

Interfaces often need to provide a way to input data: e.g. text, numbers, checkboxes, etc. In Dioxus, there are two ways you can work with user input.

Controlled Inputs

With controlled inputs, you are directly in charge of the state of the input. This gives you a lot of flexibility, and makes it easy to keep things in sync. For example, this is how you would create a controlled text input:

src/input_controlled.rs

```
pub fn App() -> Element {
    let mut name = use_signal(|| "bob".to_string());

    rsx! {
        input {
            // we tell the component what to render
            value: "{name}",
            // and what to do when the value changes
            oninput: move |event| name.set(event.value())
        }
    }
}
```



Notice the flexibility – you can:

- Also display the same contents in another element, and they will be in sync
- Transform the input every time it is modified (e.g. to make sure it is upper case)
- Validate the input every time it changes
- Have custom logic happening when the input changes (e.g. network request for autocomplete)
- Programmatically change the value (e.g. a "randomize" button that fills the input with nonsense)

Uncontrolled Inputs

As an alternative to controlled inputs, you can simply let the platform keep track of the input values. If we don't tell a HTML input what content it should have, it will be editable anyway (this is built into the browser). This approach can be more performant, but less flexible. For example, it's harder to keep the input in sync with another element.

Since you don't necessarily have the current value of the uncontrolled input in state, you can access it either by listening to `oninput` events (similarly to controlled components), or, if the input is part of a form, you can access the form data in the form events (e.g. `oninput` or `onsubmit`):

src/input_uncontrolled.rs

```
pub fn App() -> Element {
  rsx! {
    form { onsubmit: move |event| { log::info!("Submitted! {event:?}") },
      input { name: "name" }
      input { name: "age" }
      input { name: "date" }
      input { r#type: "submit" }
    }
  }
}
```

```
Submitted! UiEvent { data: FormData { value: "", values: {"age": "very old",
"date": "1966", "name": "Fred"} } }
```

Handling files

You can insert a file picker by using an input element of type `file`. This element supports the `multiple` attribute, to let you pick more files at the same time. You can select a folder by adding the `directory` attribute: Dioxus will map this attribute to browser specific attributes, because there is no standardized way to allow a directory to be selected.

`type` is a Rust keyword, so when specifying the type of the input field, you have to write it as `r#type:"file"`.

Extracting the selected files is a bit different from what you may typically use in Javascript.

The `FormData` event contains a `files` field with data about the uploaded files. This field contains a `FileEngine` struct which lets you fetch the filenames selected by the user. This example saves the filenames of the selected files to a `Vec`:

src/input_fileengine.rs

```
pub fn App() -> Element {
  let mut filenames: Signal<Vec<String>> = use_signal(Vec::new);
  rsx! {
    input {
      // tell the input to pick a file
      r#type: "file",
      // list the accepted extensions
      accept: ".txt,.rs",
      // pick multiple files
      multiple: true,
      onchange: move |evt| {
        if let Some(file_engine) = &evt.files() {
          let files = file_engine.files();
          for file_name in files {
            filenames.write().push(file_name);
          }
        }
      }
    }
  }
}
```

If you're planning to read the file content, you need to do it asynchronously, to keep the rest of the UI interactive. This example event handler loads the content of the selected files in an async closure:

src/input_fileengine_async.rs

```
onchange: move |evt| {
  async move {
    if let Some(file_engine) = evt.files() {
      let files = file_engine.files();
      for file_name in &files {
        if let Some(file) =
file_engine.read_file_to_string(file_name).await
        {
          files_uploaded.write().push(file);
        }
      }
    }
  }
}
```

Lastly, this example shows you how to select a folder, by setting the `directory` attribute to `true`.

src/input_fileengine_folder.rs

```
input {
  r#type: "file",
  // Select a folder by setting the directory attribute
  directory: true,
  onchange: move |evt| {
    if let Some(file_engine) = evt.files() {
      let files = file_engine.files();
      for file_name in files {
        println!("{}", file_name);
      }
    }
  }
}
```

Sharing State

Often, multiple components need to access the same state. Depending on your needs, there are several ways to implement this.

Lifting State

One approach to share state between components is to "lift" it up to the nearest common ancestor. This means putting the `use_signal` hook in a parent component, and passing the needed values down as props.

Suppose we want to build a meme editor. We want to have an input to edit the meme caption, but also a preview of the meme with the caption. Logically, the meme and the input are 2 separate components, but they need access to the same state (the current caption).

Of course, in this simple example, we could write everything in one component – but it is better to split everything out in smaller components to make the code more reusable, maintainable, and performant (this is even more important for larger, complex apps).

We start with a Meme component, responsible for rendering a meme with a given caption:

src/meme_editor.rs

```
#[component]
fn Meme(caption: String) -> Element {
    let container_style = r#"
        position: relative;
        width: fit-content;
    "#;

    let caption_container_style = r#"
        position: absolute;
        bottom: 0;
        left: 0;
        right: 0;
        padding: 16px 8px;
    "#;

    let caption_style = r"
        font-size: 32px;
        margin: 0;
        color: white;
        text-align: center;
    ";

    rsx! {
        div { style: "{container_style}",
            img { src: "https://i.imgflip.com/2zh47r.jpg", height: "500px" }
            div { style: "{caption_container_style}", p { style:
                "{caption_style}", "{caption}" } }
        }
    }
}
```

Note that the Meme component is unaware where the caption is coming from – it could be stored in `use_signal`, or a constant. This ensures that it is very reusable – the same component can be used for a meme gallery without any changes!

We also create a caption editor, completely decoupled from the meme. The caption editor must not store the caption itself – otherwise, how will we provide it to the Meme component? Instead, it should accept the current caption as a prop, as well as an event handler to delegate input events to:

src/meme_editor.rs

```
#[component]
fn CaptionEditor(caption: String, oninput: EventHandler<FormEvent>) -> Element {
    let input_style = r"
        border: none;
        background: cornflowerblue;
        padding: 8px 16px;
        margin: 0;
        border-radius: 4px;
        color: white;
    ";
```

```

    rsx! {
      input {
        style: "{input_style}",
        value: "{caption}",
        oninput: move |event| oninput.call(event)
      }
    }
  }
}

```

Finally, a third component will render the other two as children. It will be responsible for keeping the state and passing down the relevant props.

src/meme_editor.rs

```

fn MemeEditor() -> Element {
  let container_style = r"
    display: flex;
    flex-direction: column;
    gap: 16px;
    margin: 0 auto;
    width: fit-content;
  ";

  let mut caption = use_signal(|| "me waiting for my rust code to
compile".to_string());

  rsx! {
    div { style: "{container_style}",
      h1 { "Meme Editor" }
      Meme { caption: caption }
      CaptionEditor { caption: caption, oninput: move |event: FormEvent|
caption.set(event.value()) }
    }
  }
}

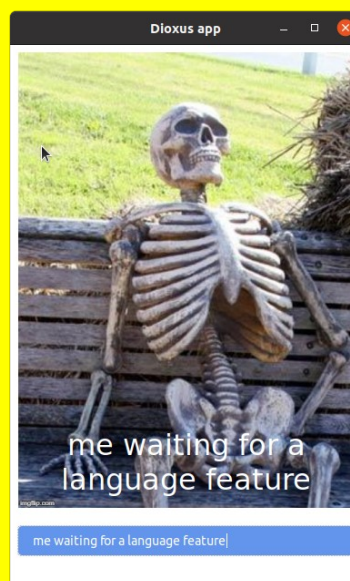
```

Using Shared State

Sometimes, some state needs to be shared between multiple components far down the tree, and passing it down through props is very inconvenient.

Suppose now that we want to implement a dark mode toggle for our app. To achieve this, we will make every component select styling depending on whether dark mode is enabled or not.

Note: we're choosing this approach for the sake of an example. There are better



ways to implement dark mode (e.g. using CSS variables). Let's pretend CSS variables don't exist – welcome to 2013!

Now, we could write another `use_signal` in the top component, and pass `is_dark_mode` down to every component through props. But think about what will happen as the app grows in complexity – almost every component that renders any CSS is going to need to know if dark mode is enabled or not – so they'll all need the same dark mode prop. And every parent component will need to pass it down to them. Imagine how messy and verbose that would get, especially if we had components several levels deep!

Dioxus offers a better solution than this "prop drilling" – providing context. The [use_context_provider](#) hook provides any `Clone` context (including `Signals`!) to any child components. Child components can use the [use_context](#) hook to get that context and if it is a `Signal`, they can read and write to it.

First, we have to create a struct for our dark mode configuration:

src/meme_editor_dark_mode.rs

```
#[derive(Clone, Copy)]
struct DarkMode(bool);
```

Now, in a top-level component (like `App`), we can provide the `DarkMode` context to all children components:

src/meme_editor_dark_mode.rs

```
use_context_provider(|| Signal::new(DarkMode(false)));
```

As a result, any child component of `App` (direct or not), can access the `DarkMode` context.

src/meme_editor_dark_mode.rs

```
let dark_mode_context = use_context::<Signal<DarkMode>>();
```

`use_context` returns `Signal<DarkMode>` here, because the `Signal` was provided by the parent. If the context hadn't been provided `use_context` would have panicked.

If you have a component where the context might or not be provided, you might want to use `try_consume_context` instead, so you can handle the `None` case. The drawback of this method is that it will not memoize the value between renders, so it won't be as efficient as `use_context`, you could do it yourself with `use_hook` though.

For example, here's how we would implement the dark mode toggle, which both reads the context (to determine what color it should render) and writes to it (to toggle dark mode):

src/meme_editor_dark_mode.rs

```
pub fn DarkModeToggle() -> Element {
    let mut dark_mode = use_context::<Signal<DarkMode>>();

    let style = if dark_mode().0 { "color:white" } else { "" };

    rsx! {
```

```

    label { style: "{style}",
            "Dark Mode"
            input {
                r#type: "checkbox",
                oninput: move |event| {
                    let is_enabled = event.value() == "true";
                    dark_mode.write().0 = is_enabled;
                }
            }
        }
    }
}

```

Dynamic Rendering

Sometimes you want to render different things depending on the state/props. With Dioxus, just describe what you want to see using Rust control flow – the framework will take care of making the necessary changes on the fly if the state or props change!

Conditional Rendering

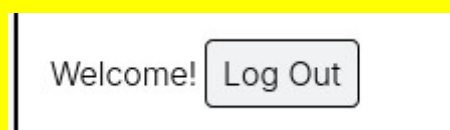
To render different elements based on a condition, you could use an `if-else` statement:

src/conditional_rendering.rs

```

if is_logged_in {
    rsx! {
        "Welcome!"
        button { onclick: move |_| log_out.call(()), "Log Out" }
    }
} else {
    rsx! {
        button { onclick: move |_| log_in.call(()), "Log In" }
    }
}

```



You could also use `match` statements, or any Rust function to conditionally render different things.

Improving the if-else Example

You may have noticed some repeated code in the `if-else` example above. Repeating code like this is both bad for maintainability and performance. Dioxus will skip diffing static elements like the button, but when switching between multiple `rsx` calls it cannot perform this optimization. For this example either approach is fine, but for components with large parts that are reused between conditionals, it can be more of an issue.

We can improve this example by splitting up the dynamic parts and inserting them where they are needed.

src/conditional_rendering.rs

```
rsx! {  
    // We only render the welcome message if we are logged in  
    // You can use if statements in the middle of a render block to  
    conditionally render elements  
    if is_logged_in {  
        // Notice the body of this if statement is rsx code, not an expression  
        "Welcome!"  
    }  
    button {  
        // depending on the value of `is_logged_in`, we will call a different  
        event handler  
        onclick: move |_| if is_logged_in { log_out.call(()) } else  
        { log_in.call(()) },  
        if is_logged_in {  
            // if we are logged in, the button should say "Log Out"  
            "Log Out"  
        } else {  
            // if we are not logged in, the button should say "Log In"  
            "Log In"  
        }  
    }  
}
```

Inspecting Element props

Since `Element` is a `Option<VNode>`, components accepting `Element` as a prop can inspect its contents, and render different things based on that. Example:

src/component_children_inspect.rs

```
fn Clickable(props: ClickableProps) -> Element {  
    rsx! {  
        div { class: "wrapper", {props.children} }  
    }  
}
```

You can't mutate the `Element`, but if you need a modified version of it, you can construct a new one based on its attributes/children/etc.

Rendering Nothing

To render nothing, you can return `NONE` from a component. This is useful if you want to conditionally hide something:

src/conditional_rendering.rs

```
if is_logged_in {  
    return rsx!();  
}  
  
rsx! {  
    p { "You must be logged in to comment" }  
}
```



You must be logged in to comment

This works because the `Element` type is just an alias for `Option<VNode>`

Again, you may use a different method to conditionally return `None`. For example the boolean's [then\(\)](#) function could be used.

Rendering Lists

Often, you'll want to render a collection of components. For example, you might want to render a list of all comments on a post.

For this, Dioxus accepts iterators that produce `Elements`. So we need to:

- Get an iterator over all of our items (e.g., if you have a `Vec` of comments, iterate over it with `iter()`)
- `.map` the iterator to convert each item into a `LazyNode` using `rsx! { ... }`
 - Add a unique key attribute to each iterator item
- Include this iterator in the final `RSX` (or use it inline)

Example: suppose you have a list of comments you want to render. Then, you can render them like this:

src/rendering_lists.rs

```
let mut comment_field = use_signal(String::new);
let mut next_id = use_signal(|| 0);
let mut comments = use_signal(Vec::<CommentData>::new);

let comments_lock = comments.read();
let comments_rendered = comments_lock.iter().map(|comment| {
    rsx! { Comment { comment: comment.clone() } }
});

rsx! {
    form {
        onsubmit: move |_| {
            comments
                .write()
                .push(CommentData {
                    content: comment_field(),
                    id: next_id(),
                });
            next_id += 1;
            comment_field.set(String::new());
        },
        input {
            value: "{comment_field}",
            oninput: move |event| comment_field.set(event.value())
        }
        input { r#type: "submit" }
    }
    {comments_rendered}
}
```

Inline for loops

Because of how common it is to render a list of items, Dioxus provides a shorthand for this. Instead of using `.iter`, `.map`, and `rsx`, you can use a `for` loop with a body of `rsx` code:

src/rendering_lists.rs

```
let mut comment_field = use_signal(String::new);
let mut next_id = use_signal(|| 0);
let mut comments = use_signal(Vec::<CommentData>::new);

rsx! {
  form {
    onsubmit: move |_| {
      comments
        .write()
        .push(CommentData {
          content: comment_field(),
          id: next_id(),
        });
      next_id += 1;
      comment_field.set(String::new());
    },
    input {
      value: "{comment_field}",
      oninput: move |event| comment_field.set(event.value())
    }
    input { r#type: "submit" }
  }
  for comment in comments() {
    // Notice the body of this for loop is rsx code, not an expression
    Comment { comment }
  }
}
```

The key Attribute

Every time you re-render your list, Dioxus needs to keep track of which items go where to determine what updates need to be made to the UI.

For example, suppose the `CommentComponent` had some state – e.g. a field where the user typed in a reply. If the order of comments suddenly changes, Dioxus needs to correctly associate that state with the same comment – otherwise, the user will end up replying to a different comment!

To help Dioxus keep track of list items, we need to associate each item with a unique key. In the example above, we dynamically generated the unique key. In real applications, it's more likely that the key will come from e.g. a database ID. It doesn't matter where you get the key from, as long as it meets the requirements:

- Keys must be unique in a list
- The same item should always get associated with the same key
- Keys should be relatively small (i.e. converting the entire `Comment` structure to a `String` would be a pretty bad key) so they can be compared efficiently

You might be tempted to use an item's index in the list as its key. That's what Dioxus will use if you don't specify a key at all. This is only acceptable if you can guarantee that the list is constant – i.e., no re-ordering, additions, or deletions.

Note that if you pass the key to a component you've made, it won't receive the key as a prop. It's only used as a hint by Dioxus itself. If your component needs an ID, you have to pass it as a separate prop.

Router

In many of your apps, you'll want to have different "scenes". For a webpage, these scenes might be the different webpages with their own content. For a desktop app, these scenes might be different views in your app.

To unify these platforms, Dioxus provides a first-party solution for scene management called Dioxus Router.

What is it?

For an app like the Dioxus landing page (<https://dioxuslabs.com>), we want to have several different scenes:

- Homepage
- Blog

Each of these scenes is independent – we don't want to render both the homepage and blog at the same time.

The Dioxus router makes it easy to create these scenes. To make sure we're using the router, add the `router` feature to your `dioxus` dependency:

```
cargo add dioxus --features router
```

Using the router

Unlike other routers in the Rust ecosystem, our router is built declaratively at compile time. This makes it possible to compose our app layout simply by defining an enum.

`src/router_reference.rs`

```
// All of our routes will be a variant of this Route enum
#[derive(Routable, PartialEq, Clone)]
enum Route {
    // if the current location is "/home", render the Home component
    #[route("/home")]
    Home {},
    // if the current location is "/blog", render the Blog component
    #[route("/blog")]
    Blog {},
}

fn Home() -> Element {
    todo!()
}

fn Blog() -> Element {
    todo!()
}
```

Whenever we visit this app, we will get either the Home component or the Blog component rendered depending on which route we enter at. If neither of these routes match the current location, then nothing will render.

We can fix this one of two ways:

- A fallback 404 page

src/router_reference.rs

```
// All of our routes will be a variant of this Route enum
#[derive(Routable, PartialEq, Clone)]
enum Route {
    #[route("/home")]
    Home {},
    #[route("/blog")]
    Blog {},
    // if the current location doesn't match any of the above routes, render
    the NotFound component
    #[route("/:..segments")]
    NotFound { segments: Vec<String> },
}

fn Home() -> Element {
    todo!()
}

fn Blog() -> Element {
    todo!()
}

#[component]
fn NotFound(segments: Vec<String>) -> Element {
    todo!()
}
```

- Redirect 404 to home

src/router_reference.rs

```
// All of our routes will be a variant of this Route enum
#[derive(Routable, PartialEq, Clone)]
enum Route {
    #[route("/home")]
    // if the current location doesn't match any of the other routes, redirect
    to "/home"
    #[redirect("/:..segments", |segments: Vec<String>| Route::Home {})]
    Home {},
    #[route("/blog")]
    Blog {},
}
```

Links

For our app to navigate these routes, we can provide clickable elements called Links. These simply wrap `<a>` elements that, when clicked, navigate the app to the given location. Because our route is an enum of valid routes, if you try to link to a page that doesn't exist, you will get a compiler error.

src/router_reference.rs

```
rsx! {  
  Link { to: Route::Home {}, "Go home!" }  
}
```

More reading

This page is just a very brief overview of the router. For more information, check out the [router book](#) or some of the [router examples](#).

Resource

[use_resource](#) lets you run an async closure, and provides you with its result.

For example, we can make an API request (using [request](#)) inside `use_resource`:

src/use_resource.rs

```
let mut future = use_resource(|| async move {  
  request::get("https://dog.ceo/api/breeds/image/random")  
    .await  
    .unwrap()  
    .json::<ApiResponse>()  
    .await  
});
```

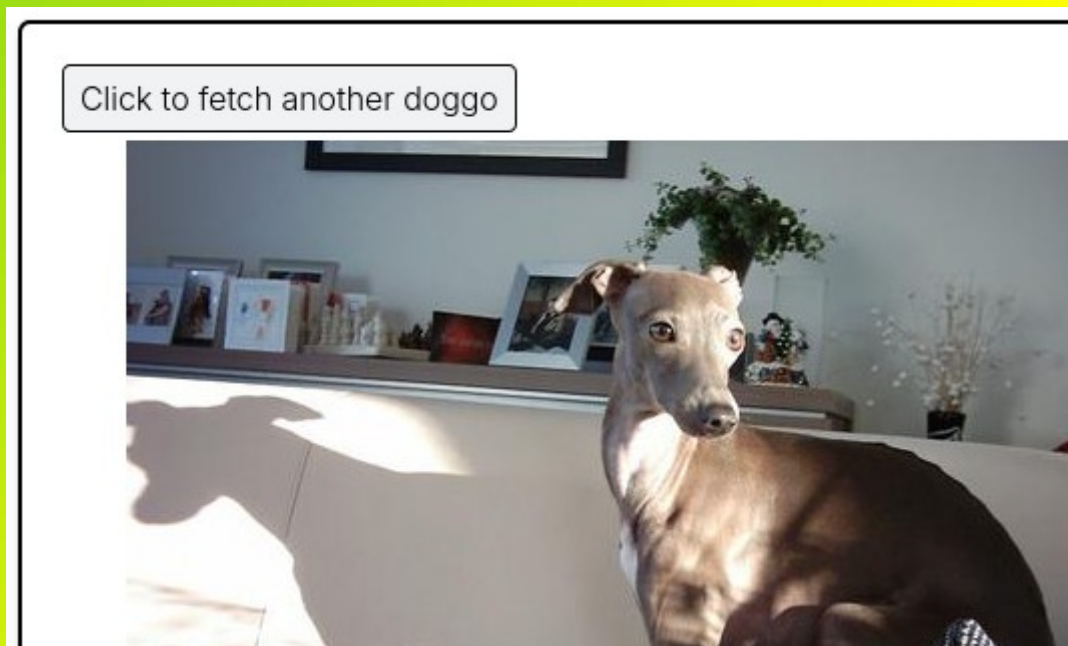
The code inside `use_resource` will be submitted to the Dioxus scheduler once the component has rendered.

We can use `&*future.read_unchecked()` to get the result of the future. On the first run, since there's no data ready when the component loads, its value will be `None`. However, once the future is finished, the component will be re-rendered and the value will now be `Some( . . . )`, containing the return value of the closure.

We can then render that result:

src/use_resource.rs

```
match &*future.read_unchecked() {  
  Some(Ok(response)) => rsx! {  
    button { onclick: move |_| future.restart(), "Click to fetch another  
doggo" }  
    div {  
      img {  
        max_width: "500px",  
        max_height: "500px",  
        src: "{response.image_url}",  
      }  
    }  
  },  
  Some(Err(_)) => rsx! {  
    div { "Loading dogs failed" }  
  },  
  None => rsx! {  
    div { "Loading dogs..." }  
  },  
}
```



Restarting the Future

The `Resource` handle provides a `restart` method. It can be used to execute the future again, producing a new value.

Dependencies

Often, you will need to run the future again every time some value (e.g. a state) changes. Rather than calling `restart` manually, you can read a signal inside of the future. It will automatically re-run the future when any of the states you read inside the future change. Example:

src/use_resource.rs

```
let future = use_resource(move || async move {
    request::get(format!("https://dog.ceo/api/breed/{breed}/images/random"))
        .await
        .unwrap()
        .json::()
        .await
});

// You can also add non-reactive state to the resource hook with the
use_reactive method
let non_reactive_state = "poodle";
use_resource(use_reactive!(|(non_reactive_state,)| async move {
    request::get(format!(
        "https://dog.ceo/api/breed/{non_reactive_state}/images/random"
    ))
        .await
        .unwrap()
        .json::()
        .await
}));
```

Coroutines

Another tool in your async toolbox are coroutines. Coroutines are futures that can have values sent to them.

Like regular futures, code in a coroutine will run until the next `await` point before yielding. This low-level control over asynchronous tasks is quite powerful, allowing for infinitely looping tasks like WebSocket polling, background timers, and other periodic actions.

use_coroutine

The `use_coroutine` hook allows you to create a coroutine. Most coroutines we write will be polling loops using `await`.

src/use_coroutine_reference.rs

```
use futures_util::StreamExt;

fn app() {
    let ws: Coroutine<()> = use_coroutine(|rx| async move {
        // Connect to some sort of service
        let mut conn = connect_to_ws_server().await;

        // Wait for data on the service
        while let Some(msg) = conn.next().await {
            // handle messages
        }
    });
}
```

For many services, a simple async loop will handle the majority of use cases.

Yielding Values

To yield values from a coroutine, simply bring in a `Signal` handle and set the value whenever your coroutine completes its work.

The future must be `'static` – so any values captured by the task cannot carry any references to `CX`, such as a `Signal`.

You can use `to_owned` to create a clone of the hook handle which can be moved into the async closure.

src/use_coroutine_reference.rs

```
let mut sync_status = use_signal(|| Status::Launching);
let sync_task = use_coroutine(move |rx: UnboundedReceiver<SyncAction>| async
move {
    loop {
        tokio::time::sleep(Duration::from_secs(1)).await;
        sync_status.set(Status::Working);
    }
});
```

To make this a bit less verbose, Dioxus exports the `to_owned!` macro which will create a binding as shown above, which can be quite helpful when dealing with many values.

src/use_coroutine_reference.rs

```
let sync_status = use_signal(|| Status::Launching);
let load_status = use_signal(|| Status::Launching);
let sync_task = use_coroutine(|rx: UnboundedReceiver<SyncAction>| {
    async move {
        // ...
    }
});
```

Sending Values

You might've noticed the `use_coroutine` closure takes an argument called `rx`. What is that? Well, a common pattern in complex apps is to handle a bunch of async code at once. With libraries like Redux Toolkit, managing multiple promises at once can be challenging and a common source of bugs.

With Coroutines, we can centralize our async logic. The `rx` parameter is an `Channel` that allows code external to the coroutine to send data *into* the coroutine. Instead of looping on an external service, we can loop on the channel itself, processing messages from within our app without needing to spawn a new future. To send data into the coroutine, we would call "send" on the handle.

src/use_coroutine_reference.rs

```
use futures_util::StreamExt;

enum ProfileUpdate {
    SetUsername(String),
    SetAge(i32),
}

let profile = use_coroutine(|mut rx: UnboundedReceiver<ProfileUpdate>| async move {
    let mut server = connect_to_server().await;

    while let Some(msg) = rx.next().await {
        match msg {
            ProfileUpdate::SetUsername(name) =>
server.update_username(name).await,
            ProfileUpdate::SetAge(age) => server.update_age(age).await,
        }
    }
});

rsx! {
    button { onclick: move |_|
profile.send(ProfileUpdate::SetUsername("Bob".to_string())),
        "Update username"
    }
}
```

Note: In order to use/run the `rx.next().await` statement you will need to extend the `[Stream]` trait (used by `[UnboundedReceiver]`) by adding 'futures_util' as a dependency to your project and adding the `use futures_util::stream::StreamExt;`

For sufficiently complex apps, we could build a bunch of different useful "services" that loop on channels to update the app.

src/use_coroutine_reference.rs

```
let profile = use_coroutine(profile_service);
let editor = use_coroutine(editor_service);
let sync = use_coroutine(sync_service);

async fn profile_service(rx: UnboundedReceiver<ProfileCommand>) {
    // do stuff
}

async fn sync_service(rx: UnboundedReceiver<SyncCommand>) {
    // do stuff
}

async fn editor_service(rx: UnboundedReceiver<EditorCommand>) {
    // do stuff
}
```

We can combine coroutines with Global State to emulate Redux Toolkit's Thunk system with much less headache. This lets us store all of our app's state *within* a task and then simply update the "view" values stored in Atoms. It cannot be understated how powerful this technique is: we get all the perks of native Rust tasks with the optimizations and ergonomics of global state. This means your *actual* state does not need to be tied up in a system like `Signal::global` or Redux – the only Atoms that need to exist are those that are used to drive the display/UI.

src/use_coroutine_reference.rs

```
static USERNAME: GlobalSignal<String> = Signal::global(||
"default".to_string());

fn app() -> Element {
    use_coroutine(sync_service);

    rsx! {
        Banner {}
    }
}

fn Banner() -> Element {
    rsx! {
        h1 { "Welcome back, {USERNAME}" }
    }
}
```

Now, in our sync service, we can structure our state however we want. We only need to update the view values when ready.

src/use_coroutine_reference.rs

```
use futures_util::StreamExt;

static USERNAME: GlobalSignal<String> = Signal::global(||
"default".to_string());
static ERRORS: GlobalSignal<Vec<String>> = Signal::global(|| Vec::new());

enum SyncAction {
    SetUsername(String),
```

```

}

async fn sync_service(mut rx: UnboundedReceiver<SyncAction>) {
    while let Some(msg) = rx.next().await {
        match msg {
            SyncAction::SetUsername(name) => {
                if set_name_on_server(&name).await.is_ok() {
                    *USERNAME.write() = name;
                } else {
                    *ERRORS.write() = vec!["Failed to set
username".to_string()];
                }
            }
        }
    }
}
}
}

```

Automatic injection into the Context API

Coroutine handles are automatically injected through the context API. You can use the `use_coroutine_handle` hook with the message type as a generic to fetch a handle.

src/use_coroutine_reference.rs

```

fn Child() -> Element {
    let sync_task = use_coroutine_handle::<SyncAction>();

    sync_task.send(SyncAction::SetUsername);

    todo!()
}

```

Spawning Futures

The `use_resource` and `use_coroutine` hooks are useful if you want to unconditionally spawn the future. Sometimes, though, you'll want to only spawn a future in response to an event, such as a mouse click. For example, suppose you need to send a request when the user clicks a "log in" button. For this, you can use `spawn`:

src/spawn.rs

```

let mut response = use_signal(|| String::from("..."));

let log_in = move |_| {
    spawn(async move {
        let resp = request::Client::new()
            .get("https://dioxuslabs.com")
            .send()
            .await;

        match resp {
            Ok(_data) => {
                log::info!("dioxuslabs.com responded!");
                response.set("dioxuslabs.com responded!".into());
            }
            Err(err) => {
                log::info!("Request failed with error: {err:?}")
            }
        }
    })
}

```

```

    });
  });
};

rsx! {
  button { onclick: log_in, "Response: {response}" }
}

```

Note: `spawn` will always spawn a *new* future. You most likely don't want to call it on every render.

Calling `spawn` will give you a `JoinHandle` which lets you cancel or pause the future.

Spawning Tokio Tasks

Sometimes, you might want to spawn a background task that needs multiple threads or talk to hardware that might block your app code. In these cases, we can directly spawn a Tokio task from our future. For Dioxus-Desktop, your task will be spawned onto Tokio's Multithreaded runtime:

src/spawn.rs

```

spawn(async {
  let _ = tokio::spawn(async {}).await;

  let _ = tokio::task::spawn_local(async {
    // some !Send work
  })
  .await;
});

```

Contributing

Development happens in the [Dioxus GitHub repository](#). If you've found a bug or have an idea for a feature, please submit an issue (but first check if someone hasn't [done it already](#)).

[GitHub discussions](#) can be used as a place to ask for help or talk about features. You can also join our [Discord channel](#) where some development discussion happens.

Improving Docs

If you'd like to improve the docs, PRs are welcome! The Rust docs ([source](#)) and this guide ([source](#)) can be found in their respective GitHub repos.

Working on the Ecosystem

Part of what makes React great is the rich ecosystem. We'd like the same for Dioxus! So if you have a library in mind that you'd like to write and many people would benefit from, it will be appreciated. You can [browse npm.js](#) for inspiration. Once you are done, add your library to the [awesome dioxus](#) list or share it in the `#I-made-a-thing` channel on [Discord](#).

Bugs & Features

If you've fixed [an open issue](#), feel free to submit a PR! Consider [reaching out](#) to the team first to make sure everyone's on the same page, and you don't do useless work!

All pull requests (including those made by a team member) must be approved by at least one other team member. Larger, more nuanced decisions about design, architecture, breaking changes, trade-offs, etc. are made by team consensus.

Before you contribute

You might be surprised that a lot of checks fail when making your first PR. That's why you should first run these commands before contributing, and it will save you *lots* of time, because the GitHub CI is much slower at executing all of these than your PC.

- Format code with [rustfmt](#):

```
cargo fmt -- src/**/*.rs
```

- You might need to install some packages on Linux (Ubuntu/deb) before the following commands will complete successfully (there is also a Nix flake in the repo root):

```
sudo apt install libgdk3.0-cil libatk1.0-dev libcairo2-dev libpango1.0-dev  
libgdk-pixbuf2.0-dev libsoup-3.0-dev libjavascriptcoregtk-4.1-dev libwebkit2gtk-  
4.1-dev
```

- Check all code [cargo check](#):

```
cargo check --workspace --examples --tests
```

- Check if [Clippy](#) generates any warnings. Please fix these!

```
cargo clippy --workspace --examples --tests -- -D warnings
```

- Test all code with [cargo-test](#):

```
cargo test --all --tests
```

- More tests, this time with [cargo-make](#). Here are all steps, including installation:

```
cargo install --force cargo-make  
cargo make tests
```

- Test with Playwright. This tests the UI itself, right in a browser. Here are all steps, including installation:**Disclaimer: This might inexplicably fail on your machine without it being your fault.** Make that PR anyway!

```
cd playwright-tests  
npm ci  
npm install -D @playwright/test  
npx playwright install --with-deps  
npx playwright test
```

How to test dioxus with local crate

If you are developing a feature, you should test it in your local setup before raising a PR. This process makes sure you are aware of your code functionality before being reviewed by peers.

- Fork the following github repo (DioxusLabs/dioxus):

<https://github.com/DioxusLabs/dioxus>

- Create a new or use an existing rust crate (ignore this step if you will use an existing rust crate): This is where we will be testing the features of the forked

```
cargo new --bin demo
```

- Add the dioxus dependency to your rust crate (new/existing) in Cargo.toml:

```
dioxus = { path = "<path to forked dioxus project>/dioxus/packages/dioxus",  
features = ["web", "router"] }
```

This above example is for dioxus-web, with dioxus-router. To know about the dependencies for different renderers visit [here](#).

- Run and test your feature

```
dx serve
```

If this is your first time with dioxus, please read [the guide](#) to get familiar with dioxus.

Project Structure

There are many packages in the Dioxus organization. This document will help you understand the purpose of each package and how they fit together:

Entry Points

- [dioxus](#): The main crate for Dioxus applications. The dioxus crate has different feature flags to enable a specific [renderer](#) with the launch API and expose different features like the router and [fullstack](#). The [CLI](#) uses the renderer feature flag that is enabled to determine what rust target to compile for.
- [dioxus-lib](#): Dioxus lib is a re-export of the dioxus crate without any renderer features. This crate is recommended for libraries because it is impossible to pull in a renderer feature accidentally which would enable that renderer for any downstream crates.

Renderers

Renderers are the entry point for Dioxus applications. They handle rendering the application, polling async tasks, and handling events. Each renderer depends on `dioxus-core` for the core virtual dom and implements both the history trait from `dioxus-history` and the event conversion trait from `dioxus-html`. Dioxus has four renderers in the main repository:

- [desktop](#): A Render that Runs Dioxus applications natively, but renders them with the system webview
- [mobile](#): A Render that Runs Dioxus applications natively, but renders them with the system webview. This is currently a think wrapper on top of the desktop renderer since both renderers use the webview
- [web](#): Renders Dioxus applications in the browser by compiling to WASM and manipulating the DOM. The web renderer has a hydration feature to take over rendering from the server if [fullstack](#) is enabled
- [liveview](#): A Render that Runs on the server, and renders using a websocket proxy in the browser. The liveview renderer is currently supported, but development has been deprioritized in favor of fullstack and it may be removed in the future

The [TUI](#) renderer has been deprecated but may be revisited in the future once the new version of Blitz is more stable

Experimental Native Rendering

In addition to the renderers listed above, Dioxus also has an experimental native renderer called Blitz that uses WebGPU to render HTML+CSS for dioxus applications:

- [taffy](#): Layout engine powering Blitz-Core, Plasmio, and Bevy UI
- [blitz](#): An experimental native renderer for Dioxus applications using WGPU

Fullstack

Fullstack can be layered on top of any renderer to add support for server functions and server-side rendering.

- [ssr](#): dioxus-ssr handles rendering a dioxus virtual dom to a string for testing or on the server. SSR is used in the fullstack renderer to handle server side rendering and static generation.
- [isrg](#): dioxus-isrg handles incremental static site generation for dioxus fullstack applications. It helps fullstack cache server side rendered routes in memory and on the file system.
- [fullstack](#): dioxus-fullstack package handles the integration between a [axum](#) server and a dioxus renderer. If the frontend renderer is targeting the web, the fullstack renderer will prepare html with embedded data so the client can take over rendering after the initial load (hydration)
- [server-macro](#): The server-macro crate defines the `server` macro used to define server functions in Dioxus applications. It integrates with the [server_fn](#) to automatically register the server functions on the server and call them on the client.

Core utilities

The core utilities contain the implementation of the virtual dom, and other macros used in all dioxus renderers. The core of dioxus doesn't assume it is running in a web context, so these utilities can be used by third party renderers like [Freyja](#).

- [core](#): The core virtual dom implementation every Dioxus application uses. The main entry point for core is the `VirtualDom`. The virtual dom diffing methods accept a cross platform

`WriteMutations` trait that is called any time the renderer need to change what is rendered. The `vdom` also has methods for running futures, and inserting events. You can read more about the architecture of the core [in this blog post](#)

- [core-types](#): The core types crate contains some of the core functions used in both in dioxus core and the hot reloading engine.
- [core-macro](#): The core macro crate implement the `derive(Props)` and `#[component]` macros to derive builds for components. It also re-exports the `rsx` macro
- [rsx](#): Implements parsing and expansion for the RSX macro. The parser is also used for hot reloading, and autoformatting in the CLI

Web utilities

Every first party dioxus renderer targets html and css. With the exception of the blitz, all renderers run inside the browser context. Dioxus has a few utilities in the workspace with shared traits and javascript bindings to help interact with the browser:

- [interpreter](#): The interpreter implements the `WriteMutations` trait from dioxus core to modify the DOM with the diffs the virtual dom generates. The interpreter is used by the desktop, web and liveview renderers. It uses a combination of [wasm-bindgen](#) and [sledgehammer-bindgen](#) to interact with the browser
- [html](#): defines html specific elements, events, and attributes. The elements and attributes are used in the `rsx` macro and hot reloading engine to map the rust identifiers to the html names. The events defined in the `html` crate are traits defined for each platform.
- [html-internal-macro](#): The `html-internal-macro` crate is used by the `html` crate to define the html elements and attributes.
- [lazy-js-bundle](#): A library to bundle typescript files at build time with bun only if the contents change. Only compiling the typescript when the files change and committing the build output lets us not require a ts compiler to be installed when dioxus is added as a library.
- [history](#): The `dioxus-history` crate defines the history trait backing each renderer must provide for use with the router. For web renderers, this should call the javascript history api. Native renderers maintain their own history stack in memory.
- [document](#): The `dioxus-document` crate defines the document trait backing each renderer must provide for use with `eval` and the `document::*` components. `eval` runs javascript code from rust, and the `document::*` components create html elements in the head.

State Management

- [generational-box](#): Generational Box is the core of all `Copy` state management in Dioxus. It allocates an arena of dynamically borrow checked values used throughout the dioxus ecosystem. The `GenerationalBox` type backs `Signal`, `Memo`, and `Resource` in dioxus signals. It is also used in `dioxus-core` to make the `Closure` and `EventHandler` types `Copy`.
- [signals](#): Signals are the main user facing state management crate for Dioxus. Signals track when they are read and written to and automatically re-run any `ReactiveContexts` that depends on the signal.

- [hooks](#): Hooks are a collection of common hooks for Dioxus applications. Most hooks are a thin wrapper over the new methods in the `signals` crate to only create the object once when the component is created.

Logging

- [logger](#): The logger crate provides a simple logging interface for Dioxus applications that works across native and wasm targets. It is automatically called in the launch function if the logging feature is enabled.

Routing

- [router](#): The router crate handles routing in Dioxus applications. It uses the history provider the renderer provides to get and modify the url. The route parsing logic is derived with the `derive(Routable)` macro defined in the `dioxus-router-macro` crate.
- [router-macro](#): The router-macro crate defines the `derive(Routable)` macro used to the route enum from a url and display it as a url.

Assets

- [manganis](#): Manganis is dioxus' asset system. It uses a macro to inject assets from rust code into the linker. Every asset gets a unique hash for cache busting. The CLI pulls the asset out of the linker and bundled them into the final application.
- [manganis-macro](#): Manganis-macro defines the `asset!( )` macro used to include assets in Dioxus applications.
- [manganis-core](#): Manganis-core contains the builders for all options passed into the `asset!( )` macro and the link sections the asset macro and CLI use to bundle assets.
- [const-serialize](#): Const Serialize defines a trait to serialize rust types to a cross platform format at compile time. This is used to serialize the options for assets at compile time in manganis.
- [const-serialize-macro](#): Const Serialize Macro defines a derive macro for types that can be serialized at compile time with the `const-serialize` crate.
- [cli-opt](#): The cli-opt optimizes the assets that manganis produces.

Formatting

- [autofmt](#): The autofmt crate finds and formats all `rsx` macros in a rust project. It uses the `dioxus-rsx` crate to parse `rsx`.

Linting

- [check](#): The dioxus-check crate analyzes dioxus code to check for common errors like calling hooks in conditionals or loops.

Translation

- [rsx-rosetta](#): The rsx-rosetta crate translates html to rsx. It uses the element definitions from `dioxus-html` to translate html elements and attributes to their rust names and the `rsx` crate to generate the rsx macro.

Hot Reloading

- [rsx-hotreload](#): The rsx-hotreload crate handles diffing rsx macros between builds and creating the hot reload templates for the CLI.
- [devtools](#): The devtools crate contains the frontend for hot reloading each renderer needs to integrate with. It receives hot reload messages from a websocket connection with the CLI
- [devtools-types](#): The devtools-types crate contains the types used to communicate between the devtools frontend and the backend in the CLI.

CLI

- [cli](#): The cli crate contains the dioxus CLI. It integrates check, autofmt, cli-opt, and rsx-hotreload to build and serve Dioxus applications.
- [cli-config](#): The cli-config crate has shared types that are provided at runtime from the CLI to crates the CLI are built with. It is used by `dioxus-desktop` to set the title from the `Dioxus.toml` file and by `dioxus-fullstack` to set the port the CLI proxies the server from.
- [dx-wire-format](#): The dx-wire-format crate has the unstable types the CLI emits in json mode. This is used by the dioxus playground.

Extension

- [extension](#): The extension folder contains the source code for the dioxus VSCode extension. It uses many of the same crates as the CLI, but packaged into a wasm+JS bundle for VSCode.

Testing

- [playwright-tests](#): The playwright-tests folder contains end to end tests for dioxus-web, dioxus-liveview and fullstack. These crates are not published on crates.io

Overall Goals

This document outlines some of the overall goals for Dioxus. These goals are not set in stone, but they represent general guidelines for the project.

The goal of Dioxus is to make it easy to build **cross-platform applications that scale**.

Cross-Platform

Dioxus is designed to be cross-platform by default. This means that it should be easy to build applications that run on the web, desktop, and mobile. However, Dioxus should also be flexible enough to allow users to opt into platform-specific features when needed. The `use_eval` is one example of this. By default, Dioxus does not assume that the platform supports JavaScript, but it does provide a hook that allows users to opt into JavaScript when needed.

Performance

As Dioxus applications grow, they should remain relatively performant without the need for manual optimizations. There will be cases where manual optimizations are needed, but Dioxus should try to make these cases as rare as possible.

One of the benefits of the core architecture of Dioxus is that it delivers reasonable performance even when components are rerendered often. It is based on a Virtual Dom which performs diffing which should prevent unnecessary re-renders even when large parts of the component tree are rerun. On top of this, Dioxus groups static parts of the RSX tree together to skip diffing them entirely.

Type Safety

As teams grow, the Type safety of Rust is a huge advantage. Dioxus should leverage this advantage to make it easy to build applications with large teams.

To take full advantage of Rust's type system, Dioxus should try to avoid exposing public `Any` types and string-ly typed APIs where possible.

Developer Experience

Dioxus should be easy to learn and ergonomic to use.

- The API of Dioxus attempts to remain close to React's API where possible. This makes it easier for people to learn Dioxus if they already know React
- We can avoid the tradeoff between simplicity and flexibility by providing multiple layers of API: One for the very common use case, one for low-level control
 - Hooks: the hooks crate has the most common use cases, but `use_hook` provides a way to access the underlying persistent value if needed.
 - The builder pattern in platform Configs: The builder pattern is used to default to the most common use case, but users can change the defaults if needed.
- Documentation:
 - All public APIs should have rust documentation
 - Examples should be provided for all public features. These examples both serve as documentation and testing. They are checked by CI to ensure that they continue to compile
 - The most common workflows should be documented in the guide

Introduction

The ✨**Dioxus CLI**✨ is a tool to get Dioxus projects off the ground.

There's no documentation for commands here, but you can see all commands using `dx --help` once you've installed the CLI! Furthermore, you can run `dx <command> --help` to get help with a specific command.

Features

- Build and pack a Dioxus project.
- Format `rsx` code.
- Hot Reload.
- Create a Dioxus project from a template repository.
- And more!

Create a Project

Once you have the Dioxus CLI installed, you can use it to create your own project!

Initializing a project

First, run the `dx new` command to create a new project.

It clones this [template](#), which is used to create dioxus apps.

You can create your project from a different template by passing the `template` argument:

```
dx new --template gh:dioxuslabs/dioxus-template
```

Next, navigate into your new project using `cd project-name`, or simply opening it in an IDE.

Make sure the WASM target is installed before running the projects. You can install the WASM target for rust using `rustup`:

```
rustup target add wasm32-unknown-unknown
```

Finally, serve your project with `dx serve`! The CLI will tell you the address it is serving on, along with additional info such as code warnings.

Configure Project

This chapter will teach you how to configure the CLI with the `Dioxus.toml` file. There's an [example](#) which has comments to describe individual keys. You can copy that or view this documentation for a more complete learning experience.

"🔒" indicates a mandatory item. Some headers are mandatory, but none of the keys inside them are. In that case, you only need to include the header, but no keys. It might look weird, but it's normal.

Structure

Each header has its TOML form directly under it.

Application 🔒

```
[application]
```

Application-wide configuration. Applies to both web and desktop.

- **asset_dir** - The directory with your static assets. The CLI will automatically copy these assets into the **out_dir** after a build/serve.

- `asset_dir = "public"`

- **sub_package** - The sub package in the workspace to build by default.

- `sub_package = "my-crate"`

Web.App 🔒

```
[web.app]
```

Web-specific configuration.

- **title** - The title of the web page.

- `# HTML title tag content`

```
title = "project_name"
```

- **base_path** - The base path to build the application for serving at. This can be useful when serving your application in a subdirectory under a domain. For example, when building a site to be served on GitHub Pages.

- `# The application will be served at domain.com/my_application/, so we need to modify the base_path to the path where the application will be served`
`base_path = "my_application"`

Web.Watcher 🔒

```
[web.watcher]
```

Development server configuration.

- **reload_html** - If this is true, the cli will rebuild the index.html file every time the application is rebuilt

- `reload_html = true`

- **watch_path** - The files & directories to monitor for changes

- `watch_path = ["src", "public"]`

- **index_on_404** - If enabled, Dioxus will serve the root page when a route is not found. *This is needed when serving an application that uses the router.* However, when serving your app using something else than Dioxus (e.g. GitHub Pages), you will have to check how to configure it on that platform. In GitHub Pages, you can make a copy of `index.html` named `404.html` in the same directory.

```
• index_on_404 = true
```

Web.Resource

```
[web.resource]
```

Static resource configuration.

- **style** - CSS files to include in your application.

```
• style = [  
  # Include from public_dir.  
  "./assets/style.css",  
  # Or some asset from online cdn.  
  "https://cdn.jsdelivr.net/npm/bootstrap/dist/css/bootstrap.css"  
]
```

- **script** - JavaScript files to include in your application.

```
• script = [  
  # Include from asset_dir.  
  "./public/index.js",  
  # Or from an online CDN.  
  "https://cdn.jsdelivr.net/npm/bootstrap/dist/js/bootstrap.js"  
]
```

Web.Resource.Dev

```
[web.resource.dev]
```

This is the same as `[web.resource]`, but it only works in development servers. For example, if you want to include a file in a `dx serve` server, but not a `dx serve --release` server, put it here.

Web.Proxy

```
[[web.proxy]]
```

Configuration related to any proxies your application requires during development. Proxies will forward requests to a new service.

- **backend** - The URL to the server to proxy. The CLI will forward any requests under the backend relative route to the backend instead of returning 404

```
• backend = "http://localhost:8000/api/"
```

This will cause any requests made to the dev server with prefix `/api/` to be redirected to the backend server at `http://localhost:8000`. The path and query parameters will be passed on as-is (path rewriting is currently not supported).

Web.https

```
[[web.https]]
```

Controls the https config for the CLI.

- **enabled** enables or disables https in the CLI
 - `enabled = true`
- **mkcert** enables or disables generating certs with the mkcert CLI
 - `mkcert = true`
- **key_path** sets the path to use for the https key
 - `key_path = "/path/to/key"`
- **cert_path** sets the path to use for the https cert
 - `cert_path = "/path/to/cert"`

Web.pre_compress

If this setting is enabled, the CLI will pre-compress the built assets in release mode with brotli. This setting is enabled by default.

```
[web]  
pre_compress = true
```

Web.wasm_opt

Controls the wasm-opt config for the CLI.

- **level** sets the level of optimization to use for wasm-opt in release builds.
 - z: optimize aggressively for size
 - s: optimize for size
 - 1: optimize for speed
 - 2: optimize for more for speed
 - 3: optimize for even more for speed
 - 4: optimize aggressively for speed (default)
 - `level = "z"`
- **debug** keep debug symbols in the wasm file even in release builds
 - `debug = true`

Bundle

```
[bundle]
```

Controls the bundling process for your application. Dioxus uses tauri-bundler under the hood. This section only includes a subset of the options available in tauri-bundler. More options can be found in the tauri-bundler [documentation](#).

- **identifier** - A unique identifier for your application (e.g., `com.dioxuslabs`).

- `identifier = "com.dioxuslabs"`

- **publisher** - The name of the entity publishing the application.

- `publisher = "DioxusLabs"`

- **icon** - Paths to icon files to be used in the bundle. Icon files must be square and 16, 24, 32, 64, or 256 pixels in size. PNG icons must have a 32 bit depth in the RGBA format. If you use a .icns file it must fit [this](#) format. The icons must include a .icns icon for macOS, .ico for Windows and .png for Linux.

- `icon = [
"icons/32x32.png",
"icons/128x128.png",
"icons/128x128@2x.png",
"icons/icon.icns",
"icons/icon.ico"
]`

- **resources** - Additional files to include in the bundle. Each asset is copied from the path and is accessible from the bundle at the same path. Any [assets](#) are automatically bundled with the installer.

- `resources = ["path/to/resource"]`

- **copyright** - Copyright information for the application.

- `copyright = "Copyright 2023 DioxusLabs"`

- **category** - The category of the application. Must be one of Business, DeveloperTool, Education, Entertainment, Finance, Game, ActionGame, AdventureGame, ArcadeGame, BoardGame, CardGame, CasinoGame, DiceGame, EducationalGame, FamilyGame, KidsGame, MusicGame, PuzzleGame, RacingGame, RolePlayingGame, SimulationGame, SportsGame, StrategyGame, TriviaGame, WordGame, GraphicsAndDesign, HealthcareAndFitness, Lifestyle, Medical, Music, News, Photography, Productivity, Reference, SocialNetworking, Sports, Travel, Utility, Video, or Weather

- `category = "Utility"`

- **short_description** - A brief description of the application.

- `short_description = "A utility application built with Dioxus"`

- **long_description** - A detailed description of the application.

- `long_description = "This application provides various utility functions..."`

- **external_bin** - Paths to external sidcar binaries to include in the bundle. These bundles may be accessed at runtime with the name of the binary (not the absolute path). **the target triple will be automatically added to the binary name before it is added to the bundle.**

- `external_bin = ["path/to/external_binary"]` # On macos, the binary at path/to/external_binary-aarch64-apple-darwin will be included in the bundle. It can be accessed at runtime with the name external_binary

Bundle.macos

```
[bundle.macos]
```

Configuration options for macOS bundles.

- **frameworks** - List of frameworks to include in the bundle.
• `frameworks = ["CoreML"]`
- **minimum_system_version** - Minimum macOS version required. (default: 10.13)
• `minimum_system_version = "10.13"`
- **license** - Path to the license file.
• `license = "LICENSE.txt"`
- **exception_domain** - Domain for exception handling. The domain must be lowercase without a port or protocol.
• `exception_domain = "mysite.com"`
- **signing_identity** - macOS signing identity.
• `signing_identity = "SIGNING IDENTITY KEYCHAIN ENTRY NAME"`
- **provider_short_name** - Provider short name for the bundle.
• `provider_short_name = "DioxusLabs"`
- **entitlements** - Path to the entitlements file.
• `entitlements = "entitlements.plist"`
- **hardened_runtime** - Whether to enable the [hardened runtime](#) in the bundle.
• `hardened_runtime = true`

Bundle.windows

```
[bundle.windows]
```

Configuration options for Windows bundles.

- **digest_algorithm** - Sets the file digest algorithm used for signing.
• `digest_algorithm = "sha-256"`
- **certificate_thumbprint** - SHA1 hash of the signing certificate.
• `certificate_thumbprint = "A1B2C3D4E5F6..."`
- **timestamp_url** - Sets the server to used for timestamping the signature.
• `timestamp_url = "http://timestamp.digicert.com"`
- **tsp** - Whether to use the time stamping protocol.
• `tsp = true`
- **icon_path** - Path to the icon for the system tray icon. (defaults to ./icons/icon.ico)
• `icon_path = "assets/icon.ico"`

- **webview_install_mode** - Installation mode for WebView2. EmbedBootstrapper: embed the WebView2 bootstrapper into the installer

```
[webview_install_mode.EmbedBootstrapper]  
silent = true
```

DownloadBootstrapper: download the WebView2 bootstrapper in the installer at runtime

```
[webview_install_mode.DownloadBootstrapper]  
silent = true
```

OfflineInstaller: Embed the WebView2 installer into the main installer

```
[webview_install_mode.OfflineInstaller]  
silent = true
```

FixedRuntime: Use a fixed path to the WebView2 runtime

```
[webview_install_mode.FixedRuntime]  
path = "path/to/runtime"
```

Skip: Does not install WebView2 as part of the installer. This will cause the application to fail if webview was not already installed

- **webview_install_mode** = "Skip"

Config example

This includes all fields, mandatory or not.

```
[application]  
  
# App name  
name = "project_name"  
  
# `build` & `serve` output path  
out_dir = "dist"  
  
# The static resource path  
asset_dir = "public"  
  
[web.app]  
  
# HTML title tag content  
title = "project_name"  
  
[web.watcher]  
  
# When watcher is triggered, regenerate the `index.html`  
reload_html = true  
  
# Which files or dirs will be monitored  
watch_path = ["src", "public"]  
  
# Include style or script assets  
[web.resource]  
  
# CSS style file  
style = []
```

```
# Javascript code file
script = []

[web.resource.dev]

# Same as [web.resource], but for development servers

# CSS style file
style = []

# JavaScript files
script = []

[[web.proxy]]
backend = "http://localhost:8000/api/"

[bundle]
identifier = "com.dioxuslabs"
publisher = "DioxusLabs"
icon = "assets/icon.png"
```

Translating existing HTML

Dioxus uses a custom format called RSX to represent the HTML because it is more concise and looks more like Rust code. However, it can be a pain to convert existing HTML to RSX. That's why Dioxus comes with a tool called `dx translate` that can automatically convert HTML to RSX!

`Dx translate` can make converting large chunks of HTML to RSX much easier! Lets try translating some of the HTML from the Dioxus homepage:

```
dx translate --raw "<div class=\"relative w-full mx-4 sm:mx-auto text-gray-600\"><div class=\"text-[3em] md:text-[5em] font-semibold dark:text-white text-ghdarkmetal font-sans py-12 flex flex-col\"><span>Fullstack, crossplatform,</span><span>lightning fast, fully typed.</span></div><h3 class=\"text-[2em] dark:text-white font-extralight text-ghdarkmetal pt-4 max-w-screen-md mx-auto\">Dioxus is a Rust library for building apps that run on desktop, web, mobile, and more.</h3><div class=\"pt-12 text-white text-[1.2em] font-sans font-bold flex flex-row justify-center space-x-4\"><a href=\"/learn/0.6/getting_started\" dioxus-prevent-default=\"onclick\" class=\"bg-red-600 py-2 px-8 hover:-translate-y-2 transition-transform duration-300\" data-dioxus-id=\"216\">Quickstart</a><a href=\"/learn/0.6/reference\" dioxus-prevent-default=\"onclick\" class=\"bg-blue-500 py-2 px-8 hover:-translate-y-2 transition-transform duration-300\" data-dioxus-id=\"214\">Read the docs</a></div><div class=\"max-w-screen-2xl mx-auto pt-36\"><h1 class=\"text-md\">Trusted by top companies</h1><div class=\"pt-4 flex flex-row flex-wrap justify-center\"><div class=\"h-12 w-40 bg-black p-2 m-4 flex justify-center items-center\"><img src=\"static/futurewei_bw.png\"></div><div class=\"h-12 w-40 bg-black p-2 m-4 flex justify-center items-center\"><img src=\"static/airbuslogo.svg\"></div><div class=\"h-12 w-40 bg-black p-2 m-4 flex justify-center items-center\"><img src=\"static/ESA_logo.svg\"></div><div class=\"h-12 w-40 bg-black p-2 m-4 flex justify-center items-center\"><img src=\"static/yclogo.svg\"></div><div class=\"h-12 w-40 bg-black p-2 m-4 flex justify-center items-center\"><img src=\"static/satellite.webp\"></div></div></div></div></div>"
```

We get the following RSX you can easily copy and paste into your code:

```
div { class: "relative w-full mx-4 sm:mx-auto text-gray-600",
  div { class: "text-[3em] md:text-[5em] font-semibold dark:text-white text-ghdarkmetal font-sans py-12 flex flex-col",
```

```

    span { "Fullstack, crossplatform," }
    span { "lightning fast, fully typed." }
  }
  h3 { class: "text-[2em] dark:text-white font-extralight text-ghdarkmetal pt-4
max-w-screen-md mx-auto",
    "Dioxus is a Rust library for building apps that run on desktop, web,
mobile, and more."
  }
  div { class: "pt-12 text-white text-[1.2em] font-sans font-bold flex flex-row
justify-center space-x-4",
    a {
      href: "/learn/0.6/getting_started",
      data_dioxus_id: "216",
      dioxus_prevent_default: "onclick",
      class: "bg-red-600 py-2 px-8 hover:-translate-y-2 transition-transform
duration-300",
      "Quickstart"
    }
    a {
      dioxus_prevent_default: "onclick",
      href: "/learn/0.6/reference",
      data_dioxus_id: "214",
      class: "bg-blue-500 py-2 px-8 hover:-translate-y-2 **transition**-
transform duration-300",
      "Read the docs"
    }
  }
}
div { class: "max-w-screen-2xl mx-auto pt-36",
  h1 { class: "text-md", "Trusted by top companies" }
  div { class: "pt-4 flex flex-row flex-wrap justify-center",
    div { class: "h-12 w-40 p-2 m-4 flex justify-center items-center",
      img { src: "/assets/static/futurewei_bw.png" }
    }
    div { class: "h-12 w-40 p-2 m-4 flex justify-center items-center",
      img { src: "/assets/static/airbuslogo.svg" }
    }
    div { class: "h-12 w-40 p-2 m-4 flex justify-center items-center",
      img { src: "/assets/static/ESA_logo.svg" }
    }
    div { class: "h-12 w-40 p-2 m-4 flex justify-center items-center",
      img { src: "/assets/static/yclogo.svg" }
    }
    div { class: "h-12 w-40 p-2 m-4 flex justify-center items-center",
      img { src: "/assets/static/satellite.webp" }
    }
  }
}
}
}

```

Usage

The `dx translate` command has several flags you can use to control your html input and rsx output.

You can use the `--file` flag to translate an HTML file to RSX:

```
dx translate --file index.html
```

Or you can use the `--raw` flag to translate a string of HTML to RSX:

```
dx translate --raw "<div>Hello world</div>"
```

Both of those commands will output the following JSX:

```
div { "Hello world" }
```

The `dx translate` command will output the JSX to stdout. You can use the `--output` flag to write the JSX to a file instead.

```
dx translate --raw "<div>Hello world</div>" --output index.rs
```

You can automatically create a component with the `--component` flag.

```
dx translate --raw "<div>Hello world</div>" --component
```

This will output the following component:

```
fn component() -> Element {  
  rsx! {  
    div { "Hello world" }  
  }  
}
```

To learn more about the different flags `dx translate` supports, run `dx translate --help`.